LINKÖPING UNIVERSITY
Department of Computer and Information Science
Software and Systems
Eric Elfving 2017-05-29

Computer examination in

TDDD38 Advanced Programming in C++

Date 2017-05-29

Time 14-19

Department IDA

Course code TDDD38

Exam code DAT1

Examiner

Klas Arvidsson (klas.arvidsson@liu.se)

Administrator

Anna Grabska Eklund, 28 2362

Teacher on call

Eric Elfving (eric.elfving@liu.se, 013-28 2419)
Will primarily answer exam questions using the student client.
Will only visit the exam rooms for system-related problems.

Allowed Aids (tillåtna hjälpmedel)

An English-* dictionary may be brought to the exam.
No other printed or electronic material are allowed.
The cppreference.com reference is available in the exam system.

Grading

The exam has a total of 25 points.
0-10 for grade U/FX
11-14 for grade 3/C
15-18 for grade 4/B
19-25 for grade 5/A.

Special instructions

- All communication with staff during the exam can be done in both English and Swedish.
- Don't log out at any time during the exam, only when you have finished.
- Given files are found in subdirectory `given_files` (write protected). The exam will be available as a pdf in this directory at the start of the exam.
- Files for examination must be submitted via the Student Client, see separate instructions (`given_files/student_client.pdf`)!
- When using standard library components, such as algorithms and containers, try to chose "best fit" regarding the problem to solve. Avoid unrelated/unnecessary computations and unnecessary data structures.
- C style coding may cause point reduction where C++ alternatives are available.
- Your code should compile. Commented out regions of non-compiling code may still give some points. Resource leaks and undefined behavior is important to fix.

Theory questions

Answers may be given in either Swedish or English. Write your answers to all theory questions in one text file called `THEORY.TXT` and submit it as `ASSIGNMENT #1`.

1. What does it mean that a type is *contextually convertible* to `bool`? [1p]
2. A *function-try-block* is sometimes good when defining constructors. Why are they especially good in that use-case? [1p]
3. In a template parameter type declaration, you can use `typename` or `class`. Does it matter which one you use? [1p]
4. Why should one avoid to *specialize* a function template? What is the, often better, alternative? [1p]
5. In which situation would one chose to declare a destructor pure virtual? [1p]

Practical questions

6. Copy the file `program6.cc` from the `given_files` directory to your working folder and make changes to your copy. Submit your answer as `ASSIGNMENT #6`. [5p]

   As you all know, fruits and berries have lots of important vitamins and minerals needed by the body. In this assignment you are to create a class hierarchy to symbolise a (very small) subset of all *botanical* fruits (which usually differ from the culinary definition of fruit). A fruit can either be dehiscent (they can open to release seeds) or indehiscent (releases seeds in some other way, such as decay or predation). In this short example, we are going to focus on the vitamin C content of fruit and create a program that calculate the total vitamin C content of some set of fruit.

   Create an abstract base class `Fruit` having the following public member functions:

   - `string name()` Gives the name of the current fruit, such as ”Apple” or ”Pea”.
   - `int vitamin_c_content()` Calculates the total content of vitamin C in the given fruit (expressed in mg).
   - `bool dehiscent()` Returns true if the fruit is dehiscent and false otherwise.
   - `int mass()` Returns the mass of the fruit.

   All fruit has a mass (expressed in grams) and a vitamin C concentration (in mg/100 gram of fruit).

   Create an abstract subclass to `Fruit` called `Berry`. A berry is by definition developed from one flower and contains all its seeds. All berries are indehiscent, other than that a berry has no special abilities that has to be implemented here.

   Create two direct subclasses to `Fruit`, `Apple` and `Pea`. Also create a subclass to `Berry`, `Lemon`. Information about the three fruits can be found in table 1.

Fruit (also string returned from `name`)	Vitamin C concentration (mg/100 g)	Dehiscent
Apple	4.6	No
Lemon	53	No
Pea	40	Yes

Table 1: Some fruit and their data

   Modify the given code according to these instructions and comments in the code. Please note that you should try to store as little data as possible.

7. Write your code on a file named `program7.cc` and submit it as `ASSIGNMENT #7`. [5p]

   There is an input data file, `life.txt`, in directory `given_files`.

   Please note that standard library components must be used whenever possible and always try to find "best fit". Avoid hand-written loops (for, while or do), use standard algorithms when suitable. If a user defined function is required, use lambda expression or a function object, not a normal function.

   The program reads input from standard input stream, `cin`. When the program is run, `cin` is to be redirected to read from a given input file:

   ```
   a.out < given_files/life.txt
   ```

   Output is to be written to the standard output stream, `cout`.

   The program shall do the following, step by step. Note, in each step, no more and no less than what is specified shall be done!

   1. read all words from the input file and store in a `std::vector` container
   2. print the number of words read (see example below)
   3. lower case all words in the container
   4. order the words in the container in ascending lexicographical (alphabetical) order
   5. remove all duplicates from the container, i.e. there shall be only one instance of each unique word left in the container
   6. print the number of unique words in the container (see example below)
   7. print all words in the container, with one space after each value (see example below)
   8. reorder the words in the container so they become ordered primarily by length (shortest words first), secondly in alphabetical order (words with equal length in alphabetic order).
   9. print all words in the container, with one space after each value (see example below)

   Example of output from the program (Note: the word outputs are not complete, ... symbolizes some left out words)

   ```
   172 words read.
   95 unique words found.

   The unique words in alphabetical order:
   a afraid after aint all am and anytime anywhere around as balloon be
   beautiful bungee can care dare dark ... take that the there to toast
   under up walk wanna want watch when will world worst would you your

   The unique words ordered by length:
   a i am as be do if in is it no of oh on so to up all and can doo fly
   fun get not one ooh out see the ... evening jumping ladders rabbits
   shivers anywhere beautiful sometimes especially superstitious
   ```

8. Copy the file `program_8.cc` from `given_files` to your working directory. The file contains a given class, named `Wrapper`, and a test program. Add your own code to this file and submit your answer as `ASSIGNMENT #8`. [5p]

   The given class is a rudimentary wrapper class for `int`, and the test program that creates a few `Wrapper` objects. Two things are to be done:

   - Wrapper shall be made into template, so any kind of values, fulfilling the requirements of `Wrapper`, can be wrapped.
   - We want to trace the creation and destruction of Wrapper objects, by letting "someone" print messages like the ones below.

   ```
   Object created: Wrapper<int> (0xffbfdda0)
   Object created: Wrapper<int> (0xffbfdda0)
   Object created: Wrapper<int> (0x28a08)
   Object created: Wrapper<double> (0xffbfdd88)
   Object destroyed: Wrapper<int> (0x28a08)
   Object created: Wrapper<char> (0xffbfdd78)
   Object destroyed: Wrapper<char> (0xffbfdd78)
   Object destroyed: Wrapper<double> (0xffbfdd88)
   Object destroyed: Wrapper<int> (0xffbfdda0)
   ```

   The hexadecimal values printed within parenthesis is the address of the object in question, which uniquely identifies the object.

   These trace printouts could easily be achieved by letting the Wrapper constructors and destructor do the printing. But, we do not want to modify the Wrapper in such an intrusive way, and we want to be able do the same for other classes, so instead we shall use the C++ idiom named *CRTP* (the Curiously Recurring Template Pattern).

   In CRTP we have a template class (B) with a template type parameter, say T, which implements the functionality we want to inject into another class (D), template or not. D is derived from B and D is also given as instantiation argument for T. This circular looking pattern may seem curious, but the fact that the class we inject into (D) and its members will be available in the member functions of the injected class (B) through the template parameter T, makes CRTP very powerful.

   Define a template class named `Object_Tracer`, which implements trace outputs, as described above, i.e. print out a message when an object is created, and print a message when an object is destroyed. Use `Object_Tracer` to inject such tracing functionality into Wrapper, using CRTP.

   The member functions of `Object_Tracer` shall be defined separately from the definition of `Object_Tracer`, i.e. not inclass, but still in the given file `program8.cc`. The type name of an object can be found by using a `type_info` object, created by a `typeid` expression, and then call the `type_info` member function `name()`. The type names to be printed, `Wrapper<int>` or `Wrapper<std::string>` will not appear without some effort, since `name()`, in g++, returns the mangled name, such as *7WrapperIiE*. To find the actual type name we need to demangle the name returned by `name()`. Include `demangle.h` and supply `demangle.cc` in the compilation. The function `demangle_name()`, declared in `demangle.h`, takes a `type_info` object as argument and returns the demangled name as a string.

9. Copy the file program9.cc to your working directory. Add your code to this file and submit your answer as `ASSIGNMENT #9`. [5p]

   Define an iterator class named `sort_insert_iterator`, which can be used to insert elements in a sequential container (e.g. a vector, a list, or a deque) in sorted order, ensuring that memory will be allocated when required. `sort_insert_iterator` shall

   - be a template with one template type parameter representing the container type.
   - fulfill the *OutputIterator* concept, i.e. support the operations listed later and have the following member type declarations:

     `value_type` ⇒ `void`

     `difference_type` ⇒ `void`

     `reference` ⇒ `void`

     `pointer` ⇒ `void`

     `iterator_category` ⇒ `std::output_iterator_tag`
   - have a constructor taking the container to be bound as argument and binding it in a suitable way

   Define a utility function named `sort_inserter()`, to ease creating `sort_insert_iterator` objects.

   `sort_inserter()` takes the container in question as argument and returns a `sort_insert_iterator` object bound to the container. Example, where `start` and `end` are iterators representing a range of int values:

```
std::vector<int> v1;
std::copy(start, end, sort_inserter(v1));
```

   Looking at the implementation of algorithm `copy`, we can find out most of what is required for `sort_insert_iterator`, which is the type of the iterator named `result`:

```
for (; first != last; ++result, ++first)
    *result = *first;
return result;
```

   - `operator++`, here used in its prefix version, shall be defined in both prefix and postfix version. The position to insert at is dependent on the value to be inserted (`*first`), so keeping track of any position in the container is not relevant. This means that `operator++` have nothing to do besides fulfilling the basic call semantics expected for `operator++`.
   - `operator=` is the operation which shall insert the value (`*first`) in sorted order into the container referred to by `result`. Values are to be ordered using `operator<`. The container is required to have the following member function, where `iterator` is an iterator type supplied by the container:

```
iterator insert(iterator, const T&);
```

   - `operator*` applied to result must return the `sort_insert_iterator` object, i.e return result itself.
   - result is passed by value to algorithm copy and returned by value, so `sort_insert_iterator` objects must be copyable.