LINKÖPING UNIVERSITY
Department of Computer and Information Science
Software and Systems
Eric Elfving 2016-08-17

Computer examination in

TDDD38 Advanced Programming in C++

Date 2016-08-17

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 primarialy 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.

Theory questions

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

1. Consider the following function. Why is this interface a bad design? (hint, think about ownership...) [1p]

```
X* compute(/* some args */)
{
    X* res = new X{};
    // do something with res
    return res;
}
```

2. What is meant by the keyword `constexpr` and why is it sometimes needed? [1p]

3. Give some short code example showing why it is important to always do member initialization in declaration order in a member initializer list. [1p]

4. Consider the following class: [1p]

```
class X
{

};
```

   Declare a user-defined conversion function to convert X into `int` as a member function.

5. Give an example of a class that can be passed to the function `foo` below: [1p]

```
template<typename T, typename U,
         template<typename, typename> class Container>
void foo(Container<T,U> const &)
{
}
```

Programming exercises

6. Copy the file named `program6.cc` to your working directory. The file contains instructions besides those below. Edit the file and submit your answer as `ASSIGNMENT #6`. [5p]

   Design a polymorphic class hierarchy for handling celestial bodies, such as stars, planets and moons. Stars and planets are regarded as two different kind of celestial bodies and, *in our universe*, a moon is a kind of planet. A planet always belongs to a star, and a moon always belongs to a planet.

   `Celestial_Body` is a base class for all other celestial body classes. `Celestial_Body` stores the name (string) and the size (double) of the celestial body. When a `Celestial_Body` object (subobject) is created, all data members shall be explicitly initialized, and thereafter never changed. Member functions:

   - `get_name()` returns the name.
   - `get_size()` returns the size.

   `Star` is a direct subclass of `Celestial_Body`, and also stores the name of the galaxy it belongs to (string). When a `Star` object is created, its name, size and the name of the galaxy it belongs to shall be explicitly initialized, and thereafter never changed. Member function:

   - `get_galaxy()` return the name of the galaxy the star belongs to.

   `Planet` is a direct subclass of `Celestial_Body`, and also stores a reference to the celestial body it belongs to (i.e., a `Star`), the planet's orbit time (double), and information about whether the planet is populated or not (bool). When a new `Planet` is created, its name, size, orbit time, and a pointer to the celestial body it belongs to shall always be explicitly initialized, and thereafter never changed. If no initial value for population is given, the default is "not populated" (false). Member functions:

   - `get_celestial_body()` return a pointer to the parent celestial body
   - `get_orbit_time()` return the orbit time
   - `is_populated()` return true if populated, otherwise false
   - `populated(bool)` takes a bool to modify the state for population.

   `Moon` is a direct subclass of `Planet`. There are no specific data stored for a `Moon` compared to a `Planet`, and no specific member functions. A moon belongs to a planet and in our universe moons may be populated.

   Besides required special member functions, no other functions than those mentioned above are allowed.

   Celestial body objects cannot be copied in any way.

   Also, see instructions in the given file!

   Design with care – keep within the given specifications!

7. Write your code on a file named program7.cc. Submit your answer as `ASSIGNMENT #7`. [5p]

   Note: For full credit, the problem must be solved using only standard library components and, only if required, user defined lambda expressions (not ordinary functions, even if that would work). Especially, no explicit repetition (for, while or do) is needed.

   This is about computing some statistical dispersion properties, namely minimum and maximum value, mean, median and population standard deviation for values entered into the program.

   - Mean is the sum of the values divided by the number of values.
   - Median is the value that you would find in the middle if the values were sorted and the number of values is odd. If the number of values is even, the median is computed as the mean of the two values found in the middle if the values were sorted.
   - Population standard deviation is computed as follows, step by step, but you may combine these steps in your solution:
     1. compute the difference between each value and mean
     2. square each difference
     3. compute the sum of the squared differences
     4. divide the sum of the squared differences by the number of values
     5. the square root of this quotient gives the population standard deviation

   Input data is always integer values.

   The program shall read input from standard input stream, `cin` until end of stream, and write output to standard output stream, `cout`.

   You are not allowed to sort the data. Minimal rearrangements necessary to compute required values is allowed.

   Output should look as the example below (given that input was 4 2 1 5 8 1 7 9 3 4 6 5):

```
min value.........: 1
max value.........: 9
mean..............: 4.58333
median............: 4.5
standard deviation: 2.49861
```

8. Copy file program8.cc to your working directory, add your own code and submit as `ASSIGNMENT #8`.[5p]

   Class `Wrapper` below is given.

```
class Wrapper {
public:
    Wrapper(const int);       // initialize value_
    void set(const int);      // set value_
    int get() const;          // access value_
    std::string str() const; // string representation of 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 that `Wrapper` can be used for any type `T` fulfilling needed requirements, e.g. that `T` have a string representation.
   2. Make it possible to set a policy for textual representation, 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:

```
Wrapper<int, Hexadecimal> hexint(4711);
cout << hexint.str();  // the string 0x1267 is printed
```

      When `str()` is called, policy `Hexadecimal` is to be used by `str()` to produce the string representation of the stored value.

      A string representation policy has one static 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 `Quoted`:

      - `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: there is a stream manipulator named `hex`. Example was shown above.
      - `Quoted` is to put citation marks around the ordinary string representation for the value, i.e. if value is `4711`, the string returned by `str()` will be `"4711"`, not just the string 4711.

```
Wrapper<int, Quoted> citint(4711);
cout << citint.str();  // "4711"
```

        If the stored type is `std::string` and the value is `foobar`, the string `"foobar"` should be returned by `str()`.

```
Wrapper<string, Quoted> citstring("foobar");
cout << citstring.str();  // "foobar"
```

   Create a main function that test the implementation fully.

9. Write your code on a file named `program9.cc` and submit your answer as `ASSIGNMENT #9`. [5p]

   Design a utility class named `Cycler`, which can be bound to a container, with elements of some type `T`, and then used to iterate over the elements of the container, cyclically, i.e., after visiting the last element in the container, the next element to be visited shall be the first element in the container.

   `Cycler` is a template with two template type parameters:

   - one parameter `T`, representing the container elements
   - one parameter `Container` representing the container, with default argument `vector<T>`

   The requirements on the container are (no need for you to check this):

   - contain elements of type `T`
   - have member iterators at least fulfilling standard forward iterator requirements
   - have a member function `begin()`, for obtaining an iterator to the first element in the container
   - have a member function `end()`, for obtaining a past-end-iterator for the container
   - have a member function `size()`, for obtaining the size of the container, i.e. the current number of element stored in the container

   `Cycler` must have:

   - a constructor for binding the container to cycle over, and initialize the `Cycler` object to refer to the first element in the container; no copy of the container is to be created.
   - a member function `next()`, which shall return a reference to the current element, and then move to the next element in the container, cyclically
   - a member function `reset()`, to reset the current position to the first element in the container
   - a member function `size()`, which returns the size of the container

   Write a full program to test `Cycler` in a good way.