LINKÖPING UNIVERSITY
Department of Computer and Information Science
Software and Systems
Eric Elfving 2018-01-08

Computer examination in

TDDD38 Advanced Programming in C++

Date 2018-01-08

Time 08-13

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.

1. (a) Give an example of a container that has *RandomAccess* iterators but non-contigious memory layout. [1p]

   (b) What are the consequences of declaring a member function `explicit` and on what type of member functions does it make sense? [1p]

   (c) Class `Derived` inherits from `Base`. `Base` has a virtual function `foo` which `Derived` overrides. What is the syntax to call the implementation of `foo` in `Base` from within `Derived`? [1p]

   (d) Given a code example in which *slicing* takes place. [1p]

   (e) What is a *fold expression* [1p]

Practical questions

2. In this exercise, your usage of the standard library containers and algorithms is in focus. Any hand-written iterations will give point deductions. Write exprissive code - use the algorithm that best describe your intentions. [5p]
C++11 added much to the language. In this exercise you are going to study the randomness of parts of the random header. Testing randomness can be hard and the test you are about to create might not be the best way of analysing it. The main goal is to generate points and plot them in a square image which should be viewable in a standard image viewer. Any cluster of points might support a theory of lack in randomness. We are using a very simple bitmap image format called PPM (or Netbpm). The PPM format is very simple. First we have the header which is P1 on one line to state that it is a bitmap image followed by one line containing the width and height of the image on one line. After the header the points are shown as 0 or 1, each line separated by a newline character. A simple image containing the letter J is available as `given_files/J.ppm`. You can view it in an image viewer or just open it in some text editor to get a feel for the format.

    1. The program is started with two commandline arguments, image size and output filename. If any of the follwing conditions are true, the program should print an error message and exit:
        - The wrong number of arguments are given
        - The image size is not an integer or outside the interval $[2, 1000]$
        - The filename doesn't end with ".ppm"
    2. Try to open the file. Exit with an error message if the file couldn't be opened.
    3. Create a vector (called `values` later on) with a total of SIZE*SIZE (this number is called `N` later on) number of integer values.
    4. Fill the vector with random values in the range $[0, SIZE)$ (non-inclusive) using a mersenne-twister engine and `uniform_int_distribution`.
    5. Create a new vector `points` containing `N/2` elements of type `pair<int, int>`.
    6. Create points from the `values` vector by using indexes $[0, N/2)$ as one coordinte and $[N/2, N)$ as the other. Store the values in `points`. The first element of `points` will then be the combination of index 0 and $N/2$ from `values` and the last will be index $N/2 - 1$ and $N - 1$.
    7. Sort `points`. It should be sorted by line and each line should be sorted by column.
    8. Remove duplicates from `points`.
    9. Create a `vector<vector<bool>>`, `image`, with SIZE number of `vector<bool>`, each of witch contain SIZE bool values. Fill `image` with the value `false`.
    10. For each point in `points`, set the corresponding index in `image` to `true`. Each `vector<bool>` in `image` represent one line in the output image.
    11. Print the string `"P1\n"` followed by the SIZE twice (separated by space) followed by newline to the output file.
    12. For each line (`vector<bool>`) in `image`, print the boolean values as integers (0 or 1) to the file. Separate the values with spaces and follow each line with a newline. It is ok to have a space before the newline character.

3. When profiling programs, you are often interested in time taken to run a specific part of the code. Often this is solved with hardware support, but in this exercise you are going to create a function object class that can measure the time taken to run a function. [5p]

   Create a template class `Profiler` that accepts some sort of callable (such as function, function object or closure type) as template argument. `Profiler` has a copy of the provided callable as well as a `Timer` object (available in `given_files/Timer.h`) as well as a counter for total time taken between all calls to the callable and a counter for number of calls. The following member functions are to be created:

   Constructor Two variants, a default constructor to value-initialize the callable and one that accepts a callable to create a copy.

   Function call operator Calls the callable and increments the total time taken (use `Timer::reset` before and after) and number of calls. Returns the return value from the function.

   calls Returns the number of calls to the callable.

   mean return the mean time taken for the function call. Behavior is undefined if function hasn't been called (no check required).

   This exercise will be graded according to the following scheme:

   2p `Profiler` has a good general structure and function call operator works for functions that has return type and lacks arguments.

   1p Function call operator also works for callables without return type

   1p Function call operator forwards arguments to given callable

   1p The argument forwarding is done correctly (keeps cv-qualifications and referenceness of arguments)

   The given code in `given_files/program3.cc` shows some possible test code based on these criteria.

4. The programming language Python has a built-in function called `enumerate` that, given some iterable range of values, creates an iterable range where each element is a value from the original range and its index in the range. In this exercise, you are going to mimic this function from python. [5p]
When finished, the following snippet should compile (as C++17)
(available as `given_files/program4.cc`):

```
std::vector values{1,3,4,6,7};
for ( auto [index, value] : enumerate(values) )
{
    std::cout << index << ": " << value;
}
```

To make this work, you are to create two template classes, one iterator class `Enumerate_Iterator` and one simple class `enumerate`. `Enumerate_Iterator` keeps track of an iterator (the type is provided as template argument) and an index internally, the index starts at 0 and the iterator is provided in the constructor. The following type declarations and operations should be supported or provided by `Enumerate_Iterator`:

Constructor Initialize the internal iterator

operator++ Increment the index and internal iterator. Should exist in both prefix and postfix versions

operator==, operator!= Comparison between `Enumerate_Iterators`. Just compare the internal iterator

operator* Dereference operator to get a pair containing the current index and a reference to the current value. Dereferencing past the end is undefined behavior (no check required)

operator-> Usually returns a pointer to a value. Since we are creating our values (pairs), this becomes a bit more tricky (don't want to return a pointer to a temporary object). Create a new type `Data_Proxy` in `Enumerate_Iterator` that contain a pair (index and reference to value) and overload `operator->` on that type as well (to return the address of the contained pair). `operator->` will be called recursively until an address is found.

Member types The standard library requires five type declarations on a iterator:

value_type The type returned by `operator*`

iterator_category `std::forward_iterator_tag`

difference_type Type of value returned when subtracting iterators. In this case it doesn't matter since we don't provide that operator, just use `std::ptrdiff_t`.

pointer Type returned from `operator->`

reference Reference to `value_type`

The class `enumerate` is very simple. It stores a reference to a container (type of container as template argument) passed as argument to the constructor and has the following members:

constructor Accepts a reference to a container and stores this as member

begin Returns an `Enumerate_Iterator` initialized with the start of the container

end Returns an `Enumerate_Iterator` initialized with the end of the container

5. The game chess has 64 squares with different pieces that move in specific ways. Every square has a file (column labeled 'A' to 'F') and a rank (row labeled 1 to 8). In this assignment, you are to create a class `Chess_Piece` that has a position (of type `Square`) and a `Movement_Behavior`. `Movement_Behavior` is an abstract class used as a base for a class hierarchy with one member function, `bool valid_move(Square start, Square end)` which return `true` if the current piece is able to move from `start` to `end`. [5p]

   Create the following subclasses to `Movement_Behavior`; `Rook_Behavior`, `Queen_Behavior` and `Pawn_Behavior`. A move is valid for a Rook if the difference in rank or file is `0`, Queen extends this to also accept moves diagonally (difference in rank and file is the same). A movement for Pawn is valid if the file is unchanged and the difference in rank is 1. The first movement of a pawn also accepts a difference of 2 in rank.

   Usually, this type of problem can be solved by templates, but there is one special rule in chess that makes it harder to use templates. Once a pawn reaches the other side it turns into a queen. This means that its behavior has to change. Add a new member function `bool should_change(Square)` to `Pawn_Behavior` that returns true if the square is in the last rank (just use 8 for this exercise).

   `Chess_Piece` has the following member functions:

   Constructor Accepts a Square and a pointer to `Movement_Behavior` and stores them as members.

   move(Square) Checks if it is valid (according to the current behavior) to move to the given Square and moves if so. If the current behavior is `Pawn_Behavior` and it has reached the other side (`should_change` returns true), the behavior is changed to `Queen_Behavior`.

   position Returns the current position.

   When you are done, the code given in `given_files/program5.cc` should compile. Overall class design is of course important in this exercise.