LINKÖPING UNIVERSITY
Department of Computer and Information Science
Software and Systems
Christoffer Holm 2026-01-13

Computer examination in
TDDD38 Advanced Programming in C++

Date 2026-01-13

Time 8-13

Department IDA

Course code TDDD38

Exam code DAT2

Examiner

Klas Arvidsson (klas.arvidsson@liu.se)

Administrator

Anna Grabska Eklund, 28 2362

Teacher on call

Christoffer Holm (christoffer.holm@liu.se)
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, except for the language section.

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 `~/Desktop/given_files` (write protected). The exam will be available as a `PDF` in this directory at the start of the exam.
- Files you want assessed must be submitted via the Student Client.
- 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.
- Questions marked as *Discussion* is meant to be answered textually (`txt` or `PDF`). The answers to these questions must be passed in separately from the code.

Available commands

`e++20` is used to compile with "all" warnings as *errors*.
`w++20` is used to compile with "all" warnings. **Recommended.**
`g++20` is used to compile *without* warnings.
`valgrind --tool=memcheck` is used to find memory leaks.

C++ reference

During the exam you will have *partial* access to `http://www.cppreference.com/` as well as a local mirror of it with a working search function, but only through the desktop icon "Web access". Do note that not everything on cppreference will be available (in particular the pages under the "Language" section will be blocked). If you are unable to access a page that should be available (it might have been blocked by mistake) then you can send a message through the exam client.

1. **If you passed the midterm test then skip this assignment (you get full points).** [5p]

   The *sliding average* is a fairly common operation in signal processing, high-frequency trading and other applications which has a need to smoothen noisy data.

   The sliding average consists of taking the average of N consecutive values in a sequence of values, meaning we sum N elements and then divide the result by N. Of course, there could be fewer than N values available at a specific time, in that case it is defined as the "normal" average for all available values.

   The sliding average is commonly recalculated every time a new data point is added by removing the old value and the adding the new one (instead of recalculating the whole sum). Specifically it can be recalculated as:

   $$\text{new sliding average} = \frac{N \cdot \text{old sliding average} + \text{new value} - \text{oldest value}}{N}. \quad (1)$$

   In this assignment you will create an iterator called `sliding_average_iterator` as class template which takes a template parameter `It` which represents another arbitrary iterator. This class stores two `It` iterators: `curr` and `prev` which represents the newest (`curr`) and oldest (`prev`) values in the sequence. Initially they refer to the same iterator. It also stores two positive integers: `N`, which represents the maximum number of values used in the average, and `count`, which represents the number of values currently used in the average. Finally there is a data member called `average` which contains the *previous* average calculated (starts as a default initialized value). This should have the same data type as the *value type* of `It`.

   This class must fulfill the requirements of an *input iterator* (`https://en.cppreference.com/w/cpp/named_req/InputIterator`). Note that the type aliases `difference_type` and `pointer` can be set to `void` and `iterator_category` to `std::input_iterator_tag`. It must support the following operators:

   - A constructor that takes an `It` iterator and a positive integer `N` which initializes all data members based on these parameters.
   - `operator*()` which returns the *current* average.
     If `count < N` then it returns `(count * average + current) / (count + 1)` where `current` is the value that `curr` refers to.
     If `count >= N` then it instead uses equation (1) to calculate the current average.
   - Both versions of `operator++()`, which updates `average` to whatever `operator*()` returns, and then steps `curr`. If `count < N` then it increments `count`, otherwise it increments `prev`.
   - `operator==()` and `operator!=()` which only compares `curr`, nothing else.

   Besides the class, a function template called `make_sliding_average_iterator()` must be implemented. This function takes an arbitrary iterator and an N value as parameters and returns corresponding `sliding_average_iterator`.

   There are testcases, with some comments to help explain the sliding average, given in `given_files/program1.cc`.

2. **Discussion:** Describe at least one major difference between STL algorithms and STL ranges which demonstrates why ranges aren't necessarily a direct replacement of algorithms. Explain a situation where you would pick algorithms over ranges. [1p]

3. Since the advent of mobile payments services such as Swish, Venmo, Apple Pay etc. it has been increasingly common for one person to front the whole bill for a group of people when, for example, eating at a resturant. It can however be problematic if people misremember how much they need to pay resulting in someone having to pay too much. [4p]

   In this assignment you will implement a program which reads a receipt (`receipt.txt`) and a number of payments (`payments.txt`) from two separate textfiles and does two things:

   1. Check whether the total sum of each file are the same. If they *don't* match, a warning should be printed.
   2. Print all payments that doesn't appear in the receipt file to `std::cout`.

   Requirement: Using more than two containers to solve this assignment will lead to point deduction (typically one container for the receipt and one container for the payments).

   Requirement: In this assignment you may not use loops, `std::for_each` or recursion. You must use STL algorithms and containers.

   Hint: The second step requires you to find the *difference* between the payments and the receipt.

   Hint: The choice of container will have a big impact on the amount of work needed to solve step two.

   Example output: For the two given files, `receipt.txt` and `payments.txt`, the sums doesn't match and has the following incorrect payments: 10, 50, 85 and 188.

4. A *stack allocator* is a simple way to implement a memory allocation scheme where memory is allocated in sequence, but is only made available again if the user deallocates the *latest* object, similar to a stack. The deallocation of objects can be scheduled by keeping track if which bytes are currently used, then when an allocation happens the object is placed after the *last* currently used byte. [6p]

   In this assignment you will implement this allocation scheme as a class template called `StackPool` that takes an `std::size_t` value called `nbytes` as a template parameter which represents how many bytes can at most be allocated simultaneously with this allocator.

   `StackPool` contains two C-arrays of size `nbytes`, one called `buffer` which contains `char` objects (this represents the data storage) and one C-array called `used` which contains `bool` values (representing whether or not corresponding byte in `buffer` is currently used by an allocated object). All values in `used` are initially set to `false`. The class also contains a variable called `head` which contains which index the next object will be allocated at in `buffer`.

   A stack allocator supports the following operations:

   - `allocate()` which is a member function template which that an arbitrary data type `T` and a variadic pack `Args` as template parameters. This function takes a function parameter pack `args` based on `Args` as function parameters.

     If `sizeof(T) bytes` fits in the remaining memory, this operation allocates a `T` object (using *placement new*) where `args` is passed along to the initialization of the `T` object.

     It then sets `sizeof(T)` values in `used`, starting at index `head`, to `true`.

     Finally it adds `sizeof(T)` to `head`.

     The memory location of the allocated `T` object is returned as a `T*`. If the object doesn't fit, it returns `nullptr`.
   - `deallocate()` which is a member function template that takes a `T*` as a function parameter. This operation calls the `T` destructor on the passed in pointer, then it calculates this objects index with `pointer - &buffer[0]`.

     All values from `index` to `index + sizeof(T)` in `used` are updated to `false`. Then it decrements `head` until either `used[head - 1]` is `true` or until `head` is zero.
   - `free_bytes()` which returns how many bytes are left. This is calculated by taking the difference between `nbytes` and `head`.

   The given testcases in `given_files/program4.cc` should work without modification and should result in *no* memory leaks.

   Requirement: You are not allowed to dynamically allocate *any* memory in this assignment.

5. **Discussion:** What are the five special member functions? Briefly explain each of them and describe what the purpose of each one is. How are these functions related to the *rule of five* and the *rule of zero*? [2p]

6. In this assignment you will implement a type trait called `element_type` which takes a data type that contains or refers to values of other types and reports what the contained type is. [5p]

   `element_type` is a `struct` template which takes an arbitrary type `T`. The result of the type trait is placed in an inner type alias called `type` (i.e. `using type = ...`). The following cases must be implemented:

   1. If `T` has an inner type alias called `value_type` then `type` is `T::value_type`.
   2. If `T` is a C-array then `type` is the type of the elements in the array.
   3. If `T` is a non-`const` pointer then `type` is the type pointed to.
   4. In every other case `type` is `void`.

   There are testcases given in `given_files/program6.cc`. These testcases must work without any modifications.

   Hint: Use partial specializations to implement case #2 and #3.

   Hint: You can either implement a concept (or use `requires`) to check whether `T` has an inner alias called `value_type`. Using this you can differentiate case #1 and #2.

   Alternatively you can create a helper function template with two overloads: one which has `T::value_type` as return type if it exists, and one overload for all other cases which has `void` as return type. Use `decltype` to extract the return type of these function overloads when setting `element_type::type` to differentate case #1 and #4.

7. **Discussion:** Identify which container is most appropriate to use if we need to add elements to the beginning and the end of a sequence while still maintaining a separate list of pointers to already existing elements. Explain why your chosen container is the most appropriate and briefly mention at least one other container and why it is less appropriate. [2p]

   Note: Clearly communicate any assumptions you make.