LINKÖPING UNIVERSITY
Department of Computer and Information Science
Software and Systems
Christoffer Holm 2025-01-15

Computer examination in **TDDD38** Advanced Programming in C++

Date 2025-01-15

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]

   In this assignment you will write a type trait that takes two template parameter packs and produces a new parameter pack which is the *interleaving* of the two parameter packs.

   In `given_files/program1.cc` the helper struct template `Pack` is given. `Pack` will be used to represent parameter packs.

   An interleaving of two packs (`Left` and `Right`) is the parameter pack where every other type comes from `Left` and every other comes from `Right`.

   Example: Suppose `Left = { int, float }` and `Right = { unsigned, double }`, then their *interleaving* would be `{ int, unsigned, float, double }`.

   If one of `Left` or `Right` is larger than the other, then we interleave elements as long as both of them have values and after that we just append the rest of the larger one to the result (see testcases for examples of this).

   To implement interleaving you must make the following partial specializations of the given class template `Interleave` (each of which defines a type alias called `type` that contains the resulting interleaving):

   1. When `Left = Pack<LeftTypes...>` and `Right = Pack<RightTypes...>` then set the type alias `type` to a `Pack` that contains the first type in `LeftTypes` followed the first type in `RightTypes`, with the interleaving of the rest of the types in `LeftTypes` and `RightTypes` appended to the end.
   2. When one of `Left` or `Right` is an empty pack then set the type alias `type` the non-empty pack.
   3. If both `Left` and `Right` are empty packs, then `type` is `Pack<>`.

   There are testcases given in `given_files/program1.cc`.

   Note: To implement specialization #1 you can use the given `Append` class template to append one `Pack<>` to another.

   Example: The way you use append is like this:

   `Append<Pack<int, float>, Pack<bool, double>>::type`

   which will be the type:

   `Pack<int, float, bool, double>`.

2. **Discussion:** What is a *fold-expression*? Demonstrate the syntax, and explain how it is related to variadic packs. Discuss how problems that are solved using fold-expressions can be solve *without* using fold-expressions. Give code examples for how to solve something *with* and *without* fold-expressions. [3p]

3. The file `given_files/products.txt` contains a list of products being sold in a store. Each product consists of a name (without any whitespace characters), a price and the stock (how many of that product is currently available in the store). There is a given `struct` in `given_files/program3.cc` that will be used to represent these products in the program. [4p]

   In this assignment you must implement the following steps:

   1. Read each product from `products.txt` into an `std::vector<Product>` called `products`.
   2. Order the vector so that it is partitioned into two parts: the first part contains all products that are in stock (i.e. products where `stock` is greater than 0), and the second part contains all products that are **not** in stock.

      Each part must be internally sorted in such a way that the most expensive product appears first and the cheapest one is last.

      Note: The two parts should still be stored in the `products` vector, so you must keep track of the boundary between the two parts.
   3. Print the total value of all products currently present in the store (i.e. take the sum of `price * stock` for each product).
   4. Print the two parts according to the expected output in `given_files/program3.cc`.

   Requirement: The only container that should be present in your program is the `products` vector, no other containers are permitted.

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

   Hint: To do IO operations, define the two operators `operator<<` and `operator>>` for `Product`. Note that these operations must reside in the `std` namespace. You can then use the stream iterators with `Product` objects.

4. **Discussion:** Discuss the differences between `std::list` and `std::deque`. What are the advantages and drawbacks of each container in comparison to each other? When would you use either of them instead of `std::vector`? [2p]

5. In this assignment you will implement an output iterator (`https://upp.gitlab-pages.liu.se/cppreference/en/cpp/named_req/OutputIterator.html`) which inserts values into a container in sorted order. The iterator is called `Sorted_Insertion_Iterator` and takes two template parameters, `Container` and `Comparator`. [5p]

   `Container` represent a container type that has an `.insert()` function which takes an iterator and a value (you do **not** need to check wheter this assumption is met).

   `Comparator` represents a callable type (i.e. a function, function object or lambda function) which has the following signature:

   ```
   bool(value_type const& left, value_type const& right)
   ```

   where `value_type` is the value type of `Contaier`. You do **not** have to check for this assumption either.

   `Sorted_Insertion_Iterator` will store a *reference* to some container of type `Container` and a `Comparator` object called `comparator` which the user optionally will specify in the constructor.

   Most operations of the iterator will do nothing, the central operation is `operator=` which will take an object of type `value_type` and insert it into the referenced container in *sorted order* (according to the comparator). If `comparator(left, right)` is `true` then `left` should appear before `right` in the sorted order.

   Note: Since the iterator should work for as many containers as possible, the insertion should be implemented as a loop which finds the first position for which `comparator()` returns `false` and insert the value at that position.

   Comparison between iterators are done by checking the *addresses* (pointers) of the containers each iterator refers to.

   You must also implement a function template called `sorted_inserter()` that takes the same template parameters as `Sorted_Insertion_Iterator`. This function should also give `Comparator` the default-type `std::less<value_type>`. This function template takes two parameters, a `Container` reference and an (optional) `Comparator` (if left out it should be set to a default-initialized `Comparator`).

   Note: You can skip the required type aliases for iterators in this assignment, however the `value_type` can be useful to have.

   There are several testcases given in `given_files/program5.cc`.

6. In this assignment you will implement a class template called `Optional` which takes one template parameter `T` which is an arbitrary data type. [4p]

   The idea is that `Optional` has two main states: either it stores a value of type `T` or it is empty. The user can then check whether the `Optional` has a value or not, and if it does have a value the user can retrieve said value.

   `Optional` must be implemented as either:

   - a *tagged union*, i.e. a class that has two union fields (a `T` field and a `bool` field which is `true` if the `Optional` is empty) stored in an anonymouse union.
   - A `char` array which is large enough to store a `T` object. Remember to use `std::launder()` when casting the pointer to a `T` pointer.

   `Optional` must also contain a `bool` value called `valid` which is only `true` if a value of type `T` is currently being stored.

   `Optional` should contain the following:

   - A default constructor to initialize an empty `Optional`.
   - A constructor that takes a `T` value and stores it in the `Optional`.
   - A copy constructor.
   - A destructor.
   - A copy assignment operator.
   - An `operator=` that takes a `T` value and assigns it to the `Optional`.
   - A `clear()` function that does nothing if the `Optional` is empty, and otherwise it calls the destructor of the stored `T` value and then makes the `Optional` empty.

     Hint: You can use `std::destroy_at()` to call a destructor in a general context.
   - A `is_valid()` function that returns `true` if the `Optional` contains a value and `false` if it is empty.
   - A `get_value()` function that returns a reference to the stored value (if the value doesn't exist then it is undefined how this function behaves).

   Note: Any usage of `std::optional` will result in zero points.

   Requirement: You may only create a *new* value inside the `Optional` if it is empty, otherwise you must assign any new values to the existing value using the assignment operator of `T`.

   Requirement: You may not allocate any memory in this assignment, you must instead use *placement new* to construct objects in already existing memory (i.e. members inside the `Optional` class).

   Requirement: There may be no memory leaks or memory issues in your solution.

   There are testcases given in `given_files/program6.cc`, they should not be modified.

7. **Discussion:** What is a *forwarding reference*? Give an example of a forwarding reference and explain how we know it is a forwarding reference and not another type of reference. [2p]

   How is `std::forward()` related to forwarding references?