LINKÖPING UNIVERSITY
Department of Computer and Information Science
Software and Systems
Christoffer Holm 2023-08-16

Computer examination in
TDDD38 Advanced Programming in C++

Date 2023-08-16

Time 8-13

Department IDA

Course code TDDD38

Exam code DAT1

Examiner

Klas Arvidsson (klas.arvidsson@liu.se)

Administrator

Anna Grabska Eklund, 28 2362

Teacher on call

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 handed 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/`, 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. *Note:* The search functionality should work, but only if you do it through cppreference. You *cannot* search on DuckDuckGo.

1. Mathematical vectors can quite easily be modelled with `std::vector`, however the calculations are a bit more combersome than what we would normally expect from a linear algebra library. In this assignment we will explore how to use `std::vector` as mathematical vectors. [4p]

   You will write a program that does the following:

   1. Read an integer `n` from `std::cin`, this represent the number of elements in our vectors.
   2. Read an `std::vector<double>` called `u` from `std::cin`. This should read *exactly* `n` elements.
   3. Read a second `std::vector<double>` called `v` from `std::cin`. This should also read *exactly* `n` elements.
   4. Calculate and print the *inner product* of the two vectors `u` and `v` (see further down for description of inner product).
   5. *Normalize* both `u` and `v` and print their new values (see further down for description of normalized vectors).

   This assignment covers STL algorithms so you may not use any loops or recursions. You must also choose appropriate algorithms and for this assignment `std::for_each` is **not** considered appropriate.

   Inner product: The inner product of two vectors is calculated by taking the sum of all pairwise multiplied elements from each vector, i.e. if we have vectors $u = (u_1, u_2, ..., u_n)$ and $v = (v_1, v_2, ..., n_n)$ then the inner product is $u \cdot v = u_1v_1 + u_2v_2 + ... + u_nv_n$.

   Length of vector: Note that the length of a mathematical vector can be calculated by taking the square root of the inner product of itself, i.e. $|u| = \sqrt{u \cdot u}$.

   Normalized vector: A vector is normalized if its length is equal to one. To normalize a vector divide each component by the vectors length.

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

   There are builtin ways to store callable objects, i.e. functions, lambdas or function objects as a singular common data type. But how does this work? In this assignment you will implement a class hierarchy which solves this problem.

   Note: You may *not* use `std::function` in this assignment.

   Create three classes:

   Callable a class template that takes one template type parameter. This class is completely empty, instead the whole implementation of `Callable` is located in a partial specialization of `Callable` for the type `Ret(Args...)`. This allows us to conveniently access the return type (`Ret`) and parameter types (`Args...`). The content of this specialization is described further down.

   Whenever `Callable` is mentioned, keep in mind that it refers to this specialization.

   Base_Implementation Is a polymorphic class which is declared and defined inside the `Callable` class. There are no data members and a single pure-virtual function called `perform` that returns `Ret` and takes `Args` as parameters.

   Implementation A class template defined inside `Callable` which takes one template type parameter `T` and inherits from `Base_Implementation`. We will assume that `T` is a callable type (i.e. a function, a function object or a lambda): this assumption should not enforced by the code or the compiler.

   `Implementation` contains a data member called `function` that is of type `T`. This data member must be initialized through a constructor, you determine what is reasonable to have here.

   `Implementation` overrides `perform` such that it calls `function` with the supplied arguments and returns the value returned from calling `function`.

   `Callable` has one data member called `impl` which is a `Base_Implementation` pointer. It also implements `Ret operator()(Args...)` which simply calls `perform` on `impl`.

   There must be a templated constructor that takes an arbitrary parameter `T` and dynamically allocates a `Implementation<T>` object and store its address in `impl`.

   `Callable` must overload `operator=` such that it takes an arbitrary object (*assumed* to be a callable object) and allocates an appropriate `Implementation` instance and stores its address in `impl`.

   Note: Don't worry about the rule of 5. No need to declare copy or move operations. However you might need a destructor to make sure memory is handled correctly.

   Note: You must make sure there are no memory leaks.

   There are a few testcases given in `given_files/program2.cc`.

3. **Discussion:** What are *glvalues* and how do they relate to *lvalues*? What are considered *glvalues* but **not** *lvalues*? What are the differences between those two types of *glvalues*? [2p]

4. Variadic packs are, in a sense, the meta programming analogue of an array of data types. However, due to the limitations of templates and compile-time calculations, it is a bit more involved than using normal arrays of values. [5p]

   In this assignment you will construct a type trait called `most_common` which contains a static `constexpr` variable called `value` and a type alias called `type`. `most_common` takes a variadic template parameter `Ts` and the idea is that `type` contains the most commonly occuring type in `Ts` and `value` contains the number of occurances of said type.

   To implement this you must first create a type trait called `occurance_count` that takes a data type `T` and a variadic pack `Ts`. This type trait contains a static `constexpr` variable `value` which is supposed to contain how many times `T` occurs in `Ts`.

   To implement `occurance_count` make two specializations:

   - One specialization when `Ts` is empty. In this case `value` is `0`.
   - One specialization where `Ts` is split into `First` and `Rest`, where `Rest` is a variadic template. Remember that you still have to include `T` as the first template parameter.

     In this specialization `value` is initialized to be the sum of `X` and the `value` of `most_common` evaluated over `Rest`, where `X` is `1` if `T` is the same type as `First` and `0` otherwise.

   Note: The first template parameter `T` is *not* part of the variadic pack, but will instead signal what type we are counting, so `T` is present in both specializations.

   We will use `occurance_count` to implement `most_common`, this is done by implementing two specializations of `most_common`:

   - When there is only one type `T` then `value` is `1` and `type` is `T`.
   - A specialization where the variadic pack is split into `T` and `Ts`. Implement two *private* static `constexpr` variables called `current` and `rest`. `current` contains the number of times `T` appears in the *complete* variadic pack (use `occurance_count` and recall that `T` is the first type in the variadic pack). `rest` is set to `most_common<Ts...>::value`.

     If `current` is greater than or equal to `rest`, then `value` is set to `current` and `type` is set to `T`, otherwise `value` is `rest` and `type` is `most_common<Ts...>::type`.

   There are testcases given in `given_files/program4.cc`. To make them work you must *either* implement an alias template `common_type_t` which is an alias for `common_type<...>::type` *OR* modify the tests so that they use `::type` directly.

5. **Discussion:** Explain when and why a virtual destructor is necessary in a polymorphic hierarchy. What types of problems might arise if the destructors are non-virtual? Explain what the problem is, why they happen and why making the destructor virtual fixes hit. [3p]

   Hint: Think about when dynamic dispatching happens (i.e. when virtual functions are called virtually).

6. One of the great benefits of streams is that we have a common interface for reading values of various types. The way streams are designed is however not the only possible way to do this. In this assignment you will explore alternatives for reading from input streams. Create a function template `read()` that takes an `std::istream` and an *lvalue* reference to some arbitrary type `T`. [4p]

   This function will read values from the stream and put them into the passed in variable. How the values are read from the stream depend on various properties of `T`:

   1. If `T` is `std::string`, then it reads an entire line using `std::getline()`.
   2. If `T` is a container that has a `push_back()` function, then it should repeatedly read values (using `operator>>`) of type `T::value_type` from the stream and insert them into the container. This should be repeated until its not possible to read any more values.
   3. If `T` is a container that has an `insert()` function, then it should repeatedly read `T::value_type` values and insert them at the *end* of the container until no more values can be read.
   4. Otherwise it should use the bultin `operator>>` to read a single value into the passed in variable.

   If there are conflicts between the cases than it should prioritize the one that appears *first* in the list above.

   Note that case #2 and case #3 are fairly similar. You don't have to make them share common parts, for these particular cases code duplication is OK.

   There are testcases given in `given_files/program6.cc`.

   Hint: Create a template class/function and use this one to create all the special cases. If you are having trouble that the compiler chooses the wrong overload, try to add extra arguments to the helper.

   Hint: You can either use SFINAE or C++20 concepts to solve this assignment.

7. Explain what *placement new* refers to. How does it differ from the normal *new* operator? Give a code example for how it is used. Is there anything in particular you have to consider when a variable constructed with placement new is destroyed? [2p]