LINKÖPING UNIVERSITY
Department of Computer and Information Science
Software and Systems
Christoffer Holm 2023-03-16

Computer examination in
TDDD38 Advanced Programming in C++

Date 2023-03-16

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

   `std::shared_ptr` is a reference counted smart pointer. What this means is that it behaves as a normal pointer, but with the added feature that it keeps track of how many other shared pointers are currently refering to the underlying data. This is achieved by making sure that the reference counter is incremented whenever the shared pointer is copied, and decremented whenever a shared pointer is destroyed. If the reference counter reaches zero then the underlying data is deallocated.

   In this assignment you will implement a (somewhat simplified) version of `std::shared_ptr` called `Shared_Pointer`.

   `Shared_Pointer` is a class template which takes a type `T` and stores a pointer to a `T` object and a *pointer* to an integer counter. **Note:** The counter variable is a *pointer* to the *shared* counter, which is done to ensure that it can be shared between multiple objects.

   There must be a function template called `make_shared()` that has template parameter `T` and takes an *arbitrary* number of parameters of arbitrary type. The function will return a `Shared_Pointer<T>` which references a newly allocated `T` object (constructed from the function parameters) with a newly allocated counter set to `1`.

   The following requirements must be met:

   - `Shared_Pointer` must implement the five special member functions. **Note:** copying should copy the pointers (not the data) and increment the counter. Moving should swap the pointers *without* modifying the counter.
   - The copy assignment operator must use the copy constructor to implement its logic.
   - There must be an `operator*` which returns a reference to the underlying data.
   - `Shared_Pointer` must have `operator->` which returns the data pointer.
   - There must be a `count()` function which returns the current reference count.
   - All the functions above should also work for `Shared_Pointer<T> const`. It should not be possible to change the underlying data if the `Shared_Pointer` is `const`.

   Note: The pointer to the counter can be `nullptr` (if we for example default initialize a `Shared_Pointer`), keep that in mind when implementing this class template.

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

2. **Discussion:** [3p]

   (a) Explain what move semantics is and discuss how it is related to *xvalues*. Include an explanation for what xvalues are and when they arise.

   (b) Describe what is meant with *ownership* and how it is related to move semantics.

3. *Reverse Polish notation* (RPN) is a special syntax for expressing mathmatical expressions. In RPN we list the operands before the operator rather than to either side of the operator. [5p]

   Examples: The expression `1 + 2` is expressed as `1 2 +` in RPN. A more complicated expression like `(1 + 2) * 3` is expressed as `1 2 + 3 *` in RPN.

   A stack-based RPN calculator utilizes a stack to implement the logic of the mathmatical expressions. When calculating an expression we go through the expression one *token* (word) at a time. If we find a number we push it to the stack. If we find an operator we pop the top two elements and apply the operator to them then pushing the result to the stack. If we find the special `.` command we pop whatever is on top of the stack and print it.

   Example: The following expression: `1 2 + 10 max .` will push `1` and then `2` to the stack. Then it will evaluate `+` by popping two elements from the stack and pushing their sum (`3` in this case). After this it finds the token `10` which is pushed to the stack. This means that we have `10` and `3` on the stack. Next it finds the operation `max` which will pop `3` and `10` from the stack and push the largest of them (`10` in this case). Finally it pop the top of the stack and then print it (since it found `.`).

   In this assignment we will implement a simple stack-based RPN calculator using polymorphisms and associative containers. This is done by implementing four classes:

   Action which is the base class. It has two pure-virtual functions: `evaluate()` which returns *nothing* and takes a reference to a stack (this is the stack where all operations will be performed) and `clone()` which returns a newly allocated copy of the object.

   Literal inherits from `Action` and stores an `int` called `value`. The `evaluate()` function pushes `value` to the passed in stack.

   Operator is a class template which stores a *callable* object named `function` of arbitrary type `Callable`. `Operator` inherits from `Action`. The `evaluate()` function will pop two values from the stack and store them in variables called `lhs` and `rhs`. It will then push the result of `function(lhs, rhs)` to the passed in stack.

   Print is a subclass of `Action` that stores a reference to an arbitrary output stream. `evaluate()` will pop the top of the stack and print the result to the stored output stream. It will also print a newline after the value.

   You also have to do the following:

   - Implement `make_literal()`, `make_print()` and `make_operator()` which are factories for the different subclasses. Note that they all must return the same type.
   - Create an appropriate container called `parse_table` which associates a token with an `Action` subclass object. `parse_table` is either global and constant *or* a static variable in the `parse_token()` function.
   - Complete the implementation of the `parse_token()` function. If the passed in `token` exists in `parse_table` then return a clone (using the virtual `clone()` function) of the object stored in the table. If the token doesn't exist in `parse_table` then convert the token to an `int` and return a new `Literal` object from that `int`.

   Requirement: There may not be any memory leaks in your implementation.

   Look at `given_files/program3.cc` for more details. You have to complete the implementation of the given code as well. You can freely modify the given program, but don't stray to far from the proposed structure.

4. **Discussion:** When is it *appropriate* to use `std::function` instead of representing a callable object as an arbitrary template parameter type? What advantages are there with using `std::function`? What disadvantages are there? [2p]

5. The `<type_traits>` library has a lot of useful trait classes. However one that is missing is `has_iterator` which checks whether a given type have iterators. In this assignment you will implement the `has_iterator` class/struct template that takes a template type parameter `T` and contains a `static constexpr bool` variable called `value` which is `true` if `T` have iterators and `false` otherwise. [4p]

   Requirement: You are **not** allowed to use C++20 features such as `requires` or `concepts`.

   Requirement: You are **not** allowed to make a special case for C-arrays, they must be covered by the common case.

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

   Hint: To detect iterators you need to utilize SFINAE, so make helper function templates `has_iterator_helper()` where you have two different versions: one if `T` has iterators and one if it doesn't. Use the overload resolution rules to induce a priority between these two cases.

   The problem is that we need to be able to detect which version is picked in order to translate it into a `bool` value. The easiest way to do this is to give the two cases different return types and then check which return type the expression `has_iterator_helper<T>()` has, this way we can detect which version was actually picked.

   Hint: There is a type trait which compares two types, use that to check the return type of `has_iterator_helper<T>()` when you initialize the `has_iterator<T>::value` constant.

6. In `data.txt` there are several datasets of integers given (one per line). In this assignment we want to calculate the averages of *each* dataset (i.e. each *line*) and then find the top `n` largest averages, where `n` is entered by the user via `std::cin`. Note that the input data are `int`s, but the averages must be `double` (since for example the dataset `1 3 7 2` has average value `3.25`). [4p]

   You must use appropriate STL algorithms for this assignment. You are not allowed to use any loops, `std::for_each` or recursion. You are not allowed to use `std::sort` either.

   Requirement: Each dataset must be represented as a `vector<int>`. To read them from the stream you must make an overload of `operator>>` inside the `std` namespace that reads a single line from a stream into a `vector<int>`. You can for example use stringstreams to make the separation of a line easier.

   ### Example execution (bold is user input)

   ```
   $ ./a.out
   5
   7
   2.8
   2
   1.75
   0.2
   ```

7. **Discussion:** Explain the two use cases of the `requires` keyword in C++20. Give code examples. [2p]