LINKÖPING UNIVERSITY
Department of Computer and Information Science
Software and Systems
Christoffer Holm 2022-08-17

Computer examination in
TDDD38 Advanced Programming in C++

Date 2022-08-17

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. `std::shared_ptr` is one of the two smart pointers that are defined in the STL. In this assignment we will explore how this smart pointer is implemented by creating our own (simplified) version called `Counted_Pointer`. **Note:** You are **not** allowed to use `std::shared_ptr` in this assignment. [6p]

   Whenever a `Counted_Pointer` is created a so called `Counted_Block` is allocated. This contains the actual object we wish to point to as well as a counter that keeps track of how many `Counted_Pointer`s currently are refering to this object. Whenever a pointer is copied this counter will increase and whenever one of these pointers are destroyed the counter will decrease. If the counter reaches zero then the `Counted_Block` is deallocated.

   `Counted_Pointer` is a class template that takes one template parameter `T` which represents the type our smart pointer *points* to. It contains an *inner struct* called `Counted_Block` that has two data members: `data` which is of type `T` and `count` which is an integer. `Counted_Block` should *not* be accessible from outside `Counted_Pointer`.

   `Counted_Pointer` stores a pointer to a `Counted_Block` called `block` and has the following member functions:

   - `operator*` which is an operator that takes no parameters and returns a reference to the `block->data`. There should be a `const` and non-`const` version of this operator.
   - `operator->` takes no parameters and returns the address (a pointer) of `block->data` which can be retrieved with either the `&` operator *or* the `std::addressof` function. There should be a `const` and non-`const` version of this operator.
   - `count()` is our way to retrieve the number of pointers currently pointing to the object. If `block` is not null this function returns `block->count`, otherwise it returns `0`.

   There must be a default constructor that sets `block` to `nullptr`. There is also a variadic constructor that takes an arbitrary number of parameters of arbitrary type. This constructor allocates a new `Counted_Block` and initializes `data` with the help of the passed in parameters, and sets `count` to `1`.

   You must also implement all the special member functions correctly. However beware that due to the variadic constructor there must be two overloads for the copy constructor:

   - `Counted_Pointer(Counted_Pointer const&)`
   - `Counted_Pointer(Counted_Pointer&)`

   Hint: Implement the non-const copy constructor by calling the const copy constructor directly. This way you don't get any code repetition.

   The copy constructor and assignment operator should increase `block->count` (as long as `block` is not null). The destructor should decrease `block->count`, and if it reaches `0` then delete `block`. The move constructor and assignment operator have no special behaviour: it should just swap the data members.

   A free (i.e. not a member) function template called `make_counted` must also be implemented. This function template takes a template paramter `T` and an arbitrary number of function parameters and returns a `Counted_Pointer<T>` object which is initialized with the passed in parameters.

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

   Requirement: You must correctly use forwarding references for the variadic constructor and `make_counted`. This is not tested in the testcases so make your own tests.

2. **Discussion:** What is overload resolution? Your answer must include the following: [3p]

   - What a candidate function is.
   - What parts of a function call is considered when finding candidate functions.
   - A full description of the process that disambiguates between multiple candidates.

3. In this assignment you will implement a syntax tree for simple boolean expressions. This syntax tree will have support for three operations: `Negation`, `And` and `Or`. Each tree will represent a boolean expression and supports evaluating and printing these expressions. [5p]

   To do this we will use dynamic polymorphism to represent each node in the syntax tree. Implement the following six classes:

   Expression Is the top-most base class for the other classes. It contains two *pure-virtual* member functions: `bool evaluate()` and `void print(std::ostream& os)`.

   Literal Represents a boolean value (either `true` or `false`). It has a `bool` data member called `value`. The `evaluate()` member function returns `value` while `print()` either prints the value of `value` as either `true` or `false` (**Note:** printing `1` or `0` is not OK).

   Negation Represents the negation of some expression. Contains a `Expression` pointer called `expression`. `print()` will print a `!` to the passed in stream, and after that will call `pretty_print()` on `expression` (see further down for details). `evaluate()` evaluates `expression` and negates the result (use `operator!`).

   Compound Is the direct base class of `And` and `Or`. This class represents an operator that can have multiple operands (arguments). It has a `std::vector` called `expressions` which contains expressions. `Compound` also introduces a *pure-virtual* member function `std::string glyph()` which returns the string representation of the operator (for example `"&&"` for `And`). This class then overrides `print()` as follows:
   Call `pretty_print()` (see further down) on each expression in `expressions` and print the result of `glyph()` *between* each expression.

   And Inherits from `Compound` and overrides `glyph()` so that it returns the string `"&&"`. Then override `evaluate()` such that it returns `true` if **all** expressions in `expressions` evaluates to `true`. Otherwise it returns `false`.

   Or Inherits from `Compound` and overrides `glyph()` so that it returns the string `"||"`. Then override `evaluate()` such that it returns `true` if **any** expression in `expressions` evaluates to `true`. Otherwise it returns `false`.

   Besides these classes a function called `pretty_print()` must be implemented. This functions takes two parameters: a `std::ostream&` and an expression. It will then *pretty print* the expression to the stream.

   Pretty print in this assignment simply means that you print the expression using their member function `print()`. However, if the expression has `Compound` as a base class, then it must add parenthesis around the expression.
   Requirement: You are *not* allowed to introduce any extra member functions in the classes to achieve this. Instead use the builtin runtime type information (RTTI) of the polymorphic classes.

   There is a partial test program given in `given_files/program3.cc`.

   Note: Issues with memory management will lead to point deductions.

4. **Discussion:** Describe both the initialization and the destruction order of data members in a class hierarchy. In what order are the constructors/destructors called? In what order are the data members initialized/destroyed? [2p]

5. It is quite common to associate things with a collection of *tags*. But just associating data with tags is usually not that beneficial: we want to be able to search through the data based on the tags. [4p]

   In this assignment you will write a program that takes a file and a set of tags and finds all entries in the file which contains the specified set of tags (see `given_files/program5.cc`).

   The file has the following format: each line represents an entry and its tags. The first word of a line is the entry and all other words are tags. The order of the tags have no significance.

   Example: Given the following file:

   ```
   entry1 a c b
   entry2 b c d
   entry3 d c a
   ```

   Now, if we search for all entries with the tag `b` we will get `entry1` and `entry2` since those are the entries that have the tag `b`. If we search for all entries that have `a` *and* `c` we will get `entry1` and `entry3` since those both have the specified tags. Notice how the order of the tags doesn't matter, the important part is which tags are associated with which entry.

   To implement this program you have to use STL Algorithms (or C++20 Ranges/Views). In this assignment it is also important to pick appropriate STL containers since the right choice will simplify the solution by quite a bit. You are not allowed to introduce any loops besides the ones in the given file. Nor are you allowed to use recursion or `std::for_each`.

   Note: there is no need to rewrite the given `read_table` function. In this function you should just fill in all the comments based on your choices. So it is OK (and even encouraged) that you leave the while-loops in `read_table` as they are. Your main focus should be on implementing the rest of the `main` function.

   In the `main` function you must implement the following steps:

   1. Remove all elements from `table` that does not have the specified `tags`. The recommended way to do is to check whether the element's tags includes all of `tags`. You can also check the intersection between `tags` and the element's tags.
   2. Extract the entry from each element.
   3. Print each entry to `std::cout` in *sorted* order.

   Think carefully about which containers to use in this assignment. Pay close attention to the requirements that your choosen algorithms have. Can you fulfill these requirements by choosing good STL containers?

   There is a beginning of the program given in `given_files/program5.cc`. The file `names.txt` contains 50 names tagged with `Male` or `Female` (or both). You can test your program by passing in this file and comparing with the examples in `given_files/program5.cc`. Note that the handling of command-line arguments and parsing of the file is given.

   Hint: Using `std::vector` will potentially make this assignment significantly harder.

6. **Discussion:** What are the performance benefits with using STL Ranges/Views rather than STL algorithms? Are there other benefits besides performance? Explain. [2p]

7. In this assignment we will write a function template called `count_bytes()` that takes an object and return how many *usable* bytes are stored in that object. What we mean with *usable* is simply that these are the bytes that we have direct access to. We do not count internal overhead for containers. [3p]

   `count_bytes` should work recursively for containers and arrays, meaning it will take the sum of `count_bytes()` called on each element. For the base case (i.e. when it is neither a container nor an array) it should instead just calculate the number of bytes in the object with `sizeof`.

   There are testcases in `given_files/program7.cc`. Note that it is not necessarily enough to fulfill all of these. The important part of this assignment is that the function works in the *general case*.

   Hint: Make two overloads, one for the base case and one for the recursive case. You might have to add additional parameters to make this work properly. Make sure that the base case is only selected if the object is not a container or array.

   Note: Containers and arrays can be written as one case, but it is OK to write it as separate cases if you find that easier.