LINKÖPING UNIVERSITY
Department of Computer and Information Science
Software and Systems
Christoffer Holm 2022-03-17

Computer examination in
TDDD38 Advanced Programming in C++

Date 2022-03-17

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::function` is a convenient way to represent all types of callable objects as one data type. One instance of `std::function` can store a function, a lambda or a function object. But how does this work? In this assignment you will implement your own (simplified) version of `std::function`. [6p]

   To do this you have to create three classes:

   Function a class template that takes one template type parameter. This class is completely empty and will only serve as a primary template.

   The whole implementation of `Function` is instead placed in a partial specialization of `Function` for the type `Ret(Args...)`, this way we can get access to the appropriate return (`Ret`) and parameter types (`Args...`). The content of this specialization is described further down.

   From this point forward every mention of `Function` refers to this specialization.

   Callable_Base Is a polymorphic inner class of `Function` (meaning it is declared and defined inside `Function`). This class has no data members and a pure-virtual function called `call` that returns `Ret` and takes `Args` as parameters.

   Callable An inner class template of `Function` that takes one template type parameter `T` and inherits from `Callable_Base`. We will assume that `T` is a callable type (i.e. a function, a function object or a lambda): this assumption is not enforced by the code.

   `Callable` contains a data member called `callback` that is of type `T`. This data member must be set with an appropriate constructor.

   `Callable` overrides `call` such that it calls `callback` with the supplied arguments and returns the value returned from calling `callback`.

   The described use of `Callable_Base` and `Callable` creates a common interface for all types of callable types, through this we will allow `Function` to store all different types as one data type.

   `Function` has one data member called `storage` which is a `Callable_Base` pointer. It also implements `Ret operator()(Args...)` which simply calls `call` on `storage`.

   There must be a constructor template that takes an arbitrary parameter `T` and dynamically allocates a `Callable<T>` and stores it in `storage`.

   Likewise `Function` must overload `operator=` such that it takes an arbitrary object (assumed to be a callable object) and allocates an appropriate `Callable` instance and assigns it to `storage`.

   Note: Make sure that there are no memory leaks in `Function`. You should *not* worry about the rule of 5, so no need to declare copy or move operations. However you might need a destructor to make sure memory is handled correctly.

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

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

3. In mathematics it is common to calculate the so called *arc length* of a curve. Analytically the arc length between two points of the function $f(x)$ is given by an integral which is in general hard to calculate accurately on a computer, so we will approximate the arc length by calculating the distance between a set of evenly distributed sample points. We will assume that the distance in the x-direction is constant between each adjacent pair of points. Let's call this distance `dx`. [4p]

   The value of `dx` and the y values of the points are given in a file that is passed to the program as a command line argument. To calculate the arc length of a given sample file, follow these steps:

   1. Open the given sample file (the filename is passed as an argument to the program, see `given_files/program3.cc`).
   2. Make sure that the file exists. If not, print an error message and abort the program.
   3. Read the value of `dx` as a `double` from the file.
   4. Fill a `std::vector<double>` called `values` with the rest of the content of the sample file. These are the y values.
   5. Calculate the difference between each consecutive pair of values in `values` and store the result in a new vector called `lengths`. **Hint:** Some algorithms that can be used here will result in the first element of `lengths` not being a difference, so keep an eye out for that.
   6. For each element `dy` in `lengths` apply the following formula: `std::sqrt(dx*dx + dy*dy)`
   7. Finally sum all the lengths to get the arc length of the given sample file. This value must be printed by the program.

   The aim of this assignment is to implement the steps above using STL algorithms (or C++20 ranges). This means that you are **not** allowed to do any type of manual iterations or recursion. The assignment can and must be solved without using `std::for_each`.

   There are a few sample files that you can test your program with:

   - `given_files/exp.txt` should result in an arc length of approximately `1.98905`.
   - `given_files/quarter_circle.txt` should have an arc length of approximately `1.55088`.

   Your calculated values should be around the given values above, up to at least 2 decimal points. If you find that your calculated arc lengths are too big, check that the algorithms you are using doesn't introduce some unwanted elements in the sum (see hint in step 5).

4. In this assignment you will explore how the interface of a stream can be used in a more abstract way than what we are used to. More specifically: you will create a simple stack class which supports the common stack operations. But instead of creating member functions for these operations we will use operator overloading to make the stack look and behave more like a stream. [5p]

   Create a class template `Stack` that takes one template parameter `T` that represent what data type is stored in the stack. `Stack` must have a constructor that allows the user to initialize the content of the stack with an arbitrary amount of `T` parameters.

   You can either use `std::initializer_list` or variadic templates to implement the constructor. If you choose to use variadic templates then the parameters *must* be taken as forwarding references.

   As mentioned earlier, the operations of the stack will be implemented as operator overloads, more specifically:

   - `operator<<` must be implemented as a member function of `Stack` and must take a `T` parameter. This operation will take the parameter and add it to the stack (think "push").
   - `operator>>` must also be implemented as a member function. It takes a reference to a `T` variable as its only parameter, and will assign the value at the top of the stack to that `T` variable. It will then remove the top value from the stack (think "pop").

   It must be possible to chain these operations, so think carefully about what return type they should have.

   Besides this, it should also be possible to implicitly convert a `Stack` to a `bool` value and a `T` value. This is done with the `operator bool` and `operator T` overloads respectively. The `bool` conversion should return `true` if the stack has elements and `false` otherwise, while the `T` overload returns the value at the top of the stack.

   Finally, the stack must be printable to a normal `std::ostream` using the "normal" `operator<<`. When printing the stack the first element to be printed should be the top of the stack and the final value should be the bottom. All elements should be separated with a space (see `given_files/program4.cc`).

   There are a few testcases for `T = int` given in `given_files/program4.cc`, but you should also add tests for other types (for example `std::string`).

   Hint: The internal storage for the stack is recommended to be a `std::vector` or `std::stack`.

5. **Discussion:** What is the difference between using parentheses when initializing a variable, and using curly braces? Give a list of what steps each initialization tries. Are there any other differences that are not covered by the steps? [2p]

6. In this assignment you will create a simple program that takes text from `std::cin` and replaces each occurance of the 3 most common words with the string `"REPLACED"`. It is important that the line structure of the text is preserved meaning each newline that was entered by the user must also occur in the output. [4p]

   Your program must implement (at least) the following steps:

   1. Represent the text with a `vector<vector<string>>` called `lines` that contains each word in each line.
   2. Read each line as a string using `std::getline` and then further split each line into words (**Hint:** use a stringstream). Each word is then inserted into the current line in `lines`. This is why we have a vector of vectors: the "outer" vector represent each line while the "inner" vector represent each word in that line.
   3. As you read each word in the text, count the number of occurances of that word. This is done by creating a separate container that keeps track of how many times each unique word occured. Each time you find a new occurance of a word you increment the counter in the container. **Think carefully about which container is appropriate here.**
   4. After that you can apply the STL algorithm `std::partial_sort_copy` to find the three most frequent words and copy them (and potentially their number of occurances) into a new `std::vector` called `common`.
   5. All occurrences in `lines` of the words stored in `common` should be replaced with the string `"REPLACED"`. This can for example be done with the STL algorithm `std::replace`.
   6. Finally, print the text again, now with the replaced words. Remember that the newlines from the original text must be preserved.

   If there are multiple valid choices for the common words then the choice is up to you. The focus of this assignment is containers, so you don't have to use STL algorithms, but it does make it easier.

   Here is an example execution of the program (italics marks user input and `<ctrl+D>` is used to signal the end of user input):

```
$ ./a.out
a b c
a d e
a e f
<ctrl+D>
== Replaced text:
REPLACED b c
REPLACED d REPLACED
REPLACED REPLACED f
```

   There is a given sample text in `given_files/lorem_ipsum.txt` that you can use as well by calling your program like this:

```
./a.out < lorem_ipsum.txt
```

   An example output is found in `given_files/program6.cc`. Note that there might be multiple different choices for which the three most common words are so your output might differ slightly.

7. **Discussion:** Describe the difference between the following containers: [2p]

   - `std::list`
   - `std::deque`

   Under what circumstances would we pick one over the other? Describe a scenario for each container where that container is prefered over both `std::vector` and the other mentioned container.

   Note: The scenarios you describe doesn't have to be overly specific, it is enough if you manage to capture the unique traits of each container.

   Hint: Think about what you can and cannot do with each container. You can also consider various properties of the iterators for each container.