LINKÖPING UNIVERSITY
Department of Computer and Information Science
Software and Systems
Christoffer Holm 2021-06-02

Computer examination in

TDDD38 Advanced Programming in C++

Date 2021-06-02

Time 14-20

Department IDA

Course code TDDD38

Exam code DAT1

Staff

Teacher on call: Christoffer Holm, 013-28 13 59 (christoffer.holm@liu.se)
Will answer questions through Microsoft Teams or E-mail.
Examiner: Klas Arvidsson, 013-28 21 46 (klas.arvidsson@liu.se)
Administrator: Anna Grabska Eklund, 013-28 23 62

Grading

The exam consists of three parts. Complete solutions/answers to part I and part II are required for a passing grade. It is also required that you have submitted to the "Examination rules" submission in Lisam, which confirms that you swear to follow the rules.

The third part is designated for higher grades. It consists of two assignments. To get grade 4 you must solve one of these assignments. To get grade 5 you need to solve both.

Communication

- You can ask questions to Christoffer Holm (christoffer.holm@liu.se) through the chat in Microsoft Teams or by E-mail.
- General information will be published when necessary in Microsoft Teams through the team called `Team_TDDD38_Exam_2021-06-02`. Be sure to check there from time to time. A suggestion would be to turn on notifications in Microsoft Teams so you don't miss any important information.
- All communication with staff during the exam can be done in both English and Swedish.
- All E-mails must be sent from your official LiU E-mail address.
- In case of emergency call the teacher on call.

Rules

- You must sit in a calm environment without any other people in the same room.
- All types of communication is forbidden, the exception being questions to the course staff.
- All forms of copying are forbidden.
- You must report any and all sources of inspiration that you use. You may use cppreference.com without citing it as a source.
- When using standard library components, such as algorithms and containers, try to choose "best fit" regarding the problem to solve. Avoid unrelated/unnecessary computations and unnecessary data structures.

- C style coding is to be avoided.
- All concepts discussed during the course are OK to use.
- Your code must compile. Commented out regions of non-compiling code are acceptable if they clearly demonstrate the idea. Write a comment describing why that piece of code is commented out.
- You must be ready to demonstrate your answers to the staff after the exam if asked to.
- Failure to follow these rules will result in a *Failed* grade.

Submission

Submission will be done through Lisam on this page:
`https://studentsubmissions.app.cloud.it.liu.se/Courses/Lisam_TDDD38_2020HT_ZA/submissions`
You can also find this page by going to `https://lisam.liu.se`, navigating to the TDDD38 course page and clicking on "Submissions" in the left-hand side menu. There your should see the following submissions:

- 2021-06-02: Examination rules
- 2021-06-02: Partial submission (16:00)
- 2021-06-02: Partial submission (18:00)
- 2021-06-02: Final submission part I
- 2021-06-02: Final submission part II
- 2021-06-02: Final submission part III

Partial submission: On the marked times you must send in the current state of all your solutions (all files). *Failure to do so within 5 minutes of the marked time will result in a failing grade.* We do not expect complete or even compiling solutions at this point.
Suggestion: Set an alarm so you don't forget.

Final submission: When you are done with the exam, you must send in your solutions through "Final submission part I" and "Final submission part II". If you have attempted Part III you must also make a submission to "Final submission part III".

- Your solution(s) to part I should be source code files (.cc, .cpp, .h, .hh, .hpp).
- Your solution to part II should be a PDF document.
- Your solution(s) to part III should be one source code file per assignment and one PDF for your answers to all the questions presented in the assignments.
- The final submission must be submitted no later than 20:00.

When you have submitted your final submission in Lisam, make sure to send *all* of your files to `christoffer.holm@liu.se` and `klas.arvidsson@liu.se` by E-mail. This includes any .doc, .docx, .odt and .txt files. The subject line must be `TDDD38: Exam 2021-06-02` .

Submitting through Lisam: Attach the files to your submission and press the submit button (it doesn't matter which one if there are multiple). You can select multiple files by holding Ctrl and clicking the files you want to attach.

You will be prompted "Do you really want to submit?". Double check that you have everything, and then press "Submit" on the popup.

You will then see a popup: "You will be redirected automatically when everything is finished" Once that has finished you will redirected to the submission page. You should also get a confirmation E-mail.

Agree to the examination rules

Before starting to work on Part I you must submit the message **"I have read and understood the rules of the examination, and I swear to follow those rules"** to the submission called "2021-06-02: Examination rules" in Lisam (see above).

Do this before starting the exam!

Part I

Introduction

This part of the exam deals with practical programming skills. You will discuss your solution to this part in part II of the exam.

Note that your code should compile on Ubuntu 18 with g++ version 7 or later with the flags: `-std=c++17 -Wall -Wextra -Wpedantic`. You can test your code on ThinLinc if you don't have access to Ubuntu 18 or g++ version 7 on your local machine.

The problem

A common abstraction in programming is to conceptually apply transformations/operations on complete lists rather than on each value in the list individually. The STL algorithms is a good example of this type of abstraction.

In this assignment we are dealing with lists where the values might be of different types. Specifically a list can consist of integers, strings and lists all mixed together.

In `list_editor.cc` two structs are implemented: `Value` and `Operation`. `Value` represents all the possible types that can be stored in a list, including the lists themselves. These are:

Integer represents a normal `int` value.

Text represents a `std::string`.

List is a list of `Value` pointers. This is the type that is operated on by the `Operation` struct (see below). So this is in some regard the "main" type of object in this assignment.

A `Value` can be copied with the supplied `clone` function and printed to an `std::ostream` with the given `print` function. The behavior of these functions varies depending on what type of value it is dealing with.

`Operation` defines different types of operations we can apply to the values that are lists. Notice that these never operate on anything but `Value` objects that are lists. There are three different types of operations defined:

Duplicate takes a list and adds a copy of each element to the end of the same list.

Filter takes a user-defined predicate (a normal function pointer in the given code) that specifies which elements should be filtered out. This means that all elements in the list for whom the predicate returns `false` will be removed.

Replace Text keeps a table of strings and associated replacements which are then applied to text values in the list. This means that all text elements in the list that can be found in the substitution table will be replaced with whatever string the table specifies.

All of this has been implemented in `list_editor.cc`, but badly. Your job is to improve this code with the modern tools and constructions discussed during the course.

The assignment

You must identify **suitable** parts of the given code that can be improved, and then demonstrate how to make those improvements. Your improvement must involve:

- A STL container **OR** two different STL algorithms
- A Lambda Function with a captured value
- Classes and Polymorphism
- A Class Template **OR** A Variadic Template

Note: It is not required that you rewrite *everything*. It is enough that you rewrite parts of the code to demonstrate your ideas and understanding.

It is up to you to show that you understand these concepts. Remember that more advanced features does not necessarily imply better code.

Note: If you have trouble showing all of these concepts in one solution, you are allowed to create different solutions based on the given code. If you do this, place each solution in its own separate file and write a comment that describe which concepts you are covering in that file.

Suggestions and hints

Suggestion: Try to quickly analyze which parts will be easier and which will be harder to rewrite and plan your time accordingly. If you want to try for higher grades our recommendation is that you are done with Part I and Part II within 3 to 4 hours.

Hint: There are a lot of comments in the code. Some of these comments contain a wishlist. These are improvements that the author would like the code to contain. You are free to use these whishlists as inspiration, but there may be other parts you wish to improve which is also OK.

Hint: Some parts may be improved by completely rewriting them. Your solution doesn't have to use code from the given file, as long as your solution performs the same (or very similar) work as the given program but in a better way.

There are more hints and suggestions in the given file.

Part II

Rules

The answer to this part must be written as a text. You need to use a program where you can insert headers, text and code examples. You can for example use Microsoft Word or OpenOffice. It is also OK to use a pure text format (for example markdown). The important part is that the formatting clearly separates headers, text and code examples (and that you can export it as a pdf). The entire text should be possible to read and understand without reading your solution to part I. This means that you have to insert relevant pieces of code from your solution into the document. You document should be around 500 to 2000 words long.

The assignment

You must answer ***ALL*** of the following questions about your solution to part I. Remember to demonstrate **suitable** usage of these concepts in each question. More advanced features does not necessarily imply better code. You **must** write one header per question.

1. Describe the class hierarchy of your solution. You should do **one** of these:
   - describe the classes and their relationships textually
   - draw a UML diagram (photos of hand drawn diagrams or digitally drawn diagrams are both OK)
2. Discuss how and why your usage of polymorphism is better than the given code. Describe the reasoning behind each virtual function, each class and the encapsulation. Discuss how these things improve the design of the program.
3. Describe a place where you used a class template or a variadic template. Discuss why that template is an improvement compared to the given code.
4. Describe how you used a lambda function and why it was an improvement. Compare lambda functions with normal functions. Explain what a captured value is and describe how and why you utilized it in your lambda function.
5. Discuss your usage of either a STL container *or* two STL algorithms. Discuss why these changes are improvements compared to the given code.

Part III

Introduction

You only have to write this part if you want a higher grade. However it can also help you compensate any potential flaws in part I and/or part II.

In this part two programming assignments are presented, each paired with a question.

- To get a grade 4 you need to solve one of the assignments.
- To get a grade 5 you need to solve both assignments.

We count a solution as solved if you have fulfilled the requirements specified in the assignment and if you have answered the question.

Write your answers to the questions in a separate document that you then submit as a PDF with your code to "2021-06-02: Final submission part III". Note that your answers can be short as long as they actually answer the question.

Note: We don't expect perfect solutions. If you are *close enough* we might still grade the assignment as solved. So if you feel that you are close to a solution you can still submit it. But if you do, make sure to write comments on what you have tried and why you think it didn't work.

Note: Any solution that doesn't compile will **not** be considered solved. **So make sure to comment out any code that causes compile errors.**

Assignment 1

A *set* is a collection of objects such that it contains no duplicates, meaning all elements are unique. In this assignment you will create a class template `Is_Set` that tells us whether or not a collection of data types makes a *set*.

For example, is `{int, float}` a *set*? Yes, because there are no duplicates. However, the following collection of data types is *not* a set: `{int, bool, int}` since there are two occurances of `int`.

`Is_Set` takes an arbitrary number of elements `Ts`, and contains a `static constexpr bool` variable called `value` that is `true` if `Ts` contains no duplicate data types, and `false` otherwise.

In order to implement `Is_Set` we need to implement a way to check whether a type `T` occurs in a variadic pack `Ts`. To do this, create a class template `Contains` that take `T` and `Ts` as template parameters. Just like `Is_Set`, `Contains` should have a `static constexpr bool` variable `value`. `Contains` must have the following specializations:

- One specialization for when `T` is the same as the first type in `Ts`. In this case `value` should be `true`.
- One specialization for when `T` is *different* from the first type in `Ts`. In this case `value` should recursively check if `T` is the same as any of the remaining types in `Ts`.
- The final specialization is when `Ts` contains no types. In this case `value` should be `false`.

The logic for these specializations is that we check `T` against every type in `Ts` and if any of them are the same as `T`, then we set `value` to `true`. Otherwise, once we reach the end of `Ts` we set `value` to `false`.

We then use `Contains` to implement two specializations for `Is_Set`:

- If the passed in variadic pack is empty, then `value` should be `true`.
- Otherwise we separate the variadic pack into `T` (the first type) and `Ts` (the rest of the variadic pack). Then:
  - Check that `Ts` does not contain the type `T` by using the `Contains` class template.
  - Make sure that `Ts` is a *set* by recursively using `Is_Set`.

  If both of these conditions hold then `value` is `true`. Otherwise set `value` to `false`.

There are a few testcases for this given in `assignment1.cc`.

Note: You are not allowed to `#include` *anything*.

Question: Explain what a fold-expression is. Are there any advantages to fold-expression compared to variadic recursion?

Assignment 2

In many languages there are *join* operations that allows us to either combine two lists or join a single value with each element in a list. An example of this would be if we joined the two lists: `{1, 2, 3, 4}` and `{'a', 'b', 'c'}`. Then we would get the list:
`{(1, 'a'), (2, 'b'), (3, 'c')}`.

Another example would be if we joined the value `5` with the `std::string "hi!"`. Then we would get the list: `{(5, 'h'), (5, 'i'), (5, '!')}`.

Unfortunately there is no such operation in C++. But you will change this! In this assignment you will create a function template `join` that takes two arbitrary values of types `T` and `U` which returns a `std::vector` of `std::pair` objects.

There are three cases to consider:

1. If both `T` and `U` are containers (meaning they have iterators) then the `join` function should return a `std::vector` that consists of pairs of the values from each position in both containers. Note that the two containers *does not* necessarily contain the same data type.

   The resulting vector should have the same length as the *shortest* of the two containers.

2. If `T` is a container but `U` is not, then the resulting `std::vector` should consists of pairs where each value from the container is paired with the non-container value of type `U`. The elements from the container is the `first` field in the pair and the singular value is the `second` field.

3. The third case is just the opposite of case #2, namely that `U` is a container while `T` is not. Therefore the second element in the pair should be the elements from the container (i.e. opposite from case #2).

Note: This function must work with any combination of containers (except C-arrays) and data types (as long as those data types are copyable).

There are testcases given in `assignment2.cc`. There is also a suggestion for how to loop over two containers with different sizes.

Hint: It is likely helpful to create a helper function `join_helper` with multiple different overloads.

Hint: Case #1 should take higher priority than case #2 and case #3.

Hint: The functions `std::vector::emplace_back` and `std::make_pair` makes it easier to create `std::pair` objects.

Question: Explain what C++20 *Concepts* are and how they could have been used to simplify the implementation of `join`.