LINKÖPING UNIVERSITY
Department of Computer and Information Science
Software and Systems
Christoffer Holm 2023-06-01

Computer examination in
TDDD38 Advanced Programming in C++

Date 2023-06-01

Time 14-19

Department IDA

Course code TDDD38

Exam code DAT2

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. The heavy machinery that are operated during large-scale production or processing of products are usually monitored quite closely by various software systems. The purpose of these systems are to detect faults and problems during production. Generally these systems monitors certain parameters and ensures that most products have values that fall within some tolerance interval. [4p]

   In this assignment you must use STL algorithms to implement a program that takes a sequence of products and reports whether there are any potential problems with the production line.

   The file `factory.txt` is the log of a sequence of products processed at a production line. Each product is represented using the following format:

   ```
   <name of product> <weight of product> <volume of product>
   ```

   Where the weight is expressed in kg and volume is expressed in m^3. Note that newlines have no meaning for the format itself, those are just there to make the file easier to read for humans.

   The idea is that a product is *potentially* faulty if its density falls outside of a specified *tolerance* range. For example, if the density tolerance is:

   $$\left[1000\ \text{kg/m}^3, 1500\ \text{kg/m}^3\right]$$

   then any product with a density outside of this interval is *potentially* a fault.

   The production line however has a certain fault threshold n, where we only abort the production if the system detects n **consecutive** products that fall outside of the density tolerance (see output examples in `given_files/program1.cc` for examples of this).

   The user specifies the filename, the lower and upper tolerance as well as the fault threshold as arguments to the program (i.e. as elements `1` to `4` in `argv`). Note that the tolerance bounds are of type `double` while the threshold n is an integer type.

   In `given_files/program1.cc` there is a struct called `Product` given which is used to represent a product in the production line. `Product` also have a member function `density()` which calculates the density of the product. There are also several examples of what the output of the program should be given different inputs.

   Requirement: You are not allowed to write any manual iteration statements, nor are you allowed to use recursion. Instead you must solve all problems using STL algorithms and other appropriate components from the standard library.

   Hint: To properly read the file using only standard library components, it is a good idea to define an input stream operator (`operator>>`) for the given `Product` struct. It might also be helpful to define `operator<<`.

   Hint: Either a product falls within the acceptable tolerance or it doesn't. The idea is that you must search for n consecutive products that falls outside of the tolerance.

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

Most of us have had the need to draw various plots at some point. Drawing nice looking diagrams and plots is quite hard to do yourself and involves a lot of graphical programming. In this assignment you will use inheritance and dynamic polymorphism to implement functionality to "draw" plots directly in the terminal (see example output in `given_files/program2.cc`). Note that the plots we will draw in this assignment are transposed (meaning the x-axis grows downwards and the y-axis grows to the right) since that is much easier to print in a terminal. In this assignment we will only plot positive values.

You need to implement the following classes:

Plot is the base class of our hierarchy, all other classes inherits from this one. It represents an arbitrary plot. It has two *pure-virtual* functions, `print()` and `get_width()`. It has no data members.

`get_width()` calculates how many characters in width the printed plot is, i.e. this function should return the width of the bounding rectangle around the plot.

`print()` takes an `std::ostream` and prints the plot to that stream.

Bar_Plot Represents a bar graph/plot. It stores a `std::vector` containing `std::string` and `unsigned` pairs. These pairs represent each bar, where the string is the label and the `unsigned` represents the size.

`get_width()` should find the size of the longest label, and the size of the largest bar and then return their sum plus 1.

`print()` first finds the length of the longest label, and then it prints each bar, one bar per line. See the example output in `given_files/program2.cc` for details.

Function_Plot takes a *callable object* (either as template or using `std::function`). This callable object should adhere to the following signature: `unsigned(unsigned)`. The idea is that this class will draw a function graph by using the callable object. This class stores the callable object, as well as the lower and upper input values that will be plotted. If no bounds are given, the default range should be $[0, 10]$.

`get_width()` calculates the maximum value achieved by the callable object within the specified bounds.

`print()` iterates each integer value in the specified range. It prints a number of spaces equal to the result of the callable object, followed by a single `'+'` character.

Gallery is a gallery of multiple plots (stored in a `std::vector`). This class *owns* multiple plots and print them side-by-side (see second plot in example output).

A suggested implementation of `print()` is given in `given_files/program2.cc`. The `get_width()` function should return the sum of the widths of all its stored plots.

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

Modify the given main program in `given_files/program2.cc` so that it works properly with polymorphism without slicing.

Hint: You can use `std::initializer_list` to implement the constructors of all the classes.

3. **Discussion:** Suppose we have a class `My_Integer` which represents a custom integer type. Discuss how we can implement *implicit type conversions* between `int` and `My_Integer`. How is this related to the keyword `explicit`? [2p]

4. In this assignment you will implement two classes that will allow a user to split a container into evenly sized blocks (with the final block being potentially smaller). The idea is that code like this should work: [5p]

```
std::vector<int> v { ... };
for (auto block : make_blocks<4>(v))
{
  // note that 'block' is a vector
  for (auto element : block)
    std::cout << element << " ";
  std::cout << std::endl;
}
```

Where `v` is split into blocks of size 4 and then each block is printed on its own line.

The way we achieve this is by implementing two class templates, `Block_Container` and `Block_Iterator`. We also have to define the function template called `make_blocks()`.

All of these templates take two template parameters, `Container` which represents the type of the container we are splitting into blocks, and the non-type template parameter `block_size` which represents how many elements should be in each block. Note that these template parameters doesn't necessarily have the same *order* in each template.

Below is a description of each template:

make_blocks() is a helper function which allows us to construct a `Block_Container` in a nice way. This function takes a container reference as a parameter and constructs a `Block_Container` object from that container.

Block_Container stores a reference to a container and has two member functions: `begin()` and `end()`, both of which return appropriate `Block_Iterator` objects. `end()` should return a `Block_Iterator` where all three of its members are the *end* iterator of the underlying container (see below).

Block_Iterator is an iterator which stores three `Container::iterator` objects, namely: `block_begin`, `block_end` and `container_end`.

In the `operator++()` operators, the iterator will move from one block (the current) to the next block in the container. This is achieved by setting `block_begin` to `block_end` and by incrementing `block_end` `block_size` times *OR* until it reaches `container_end`. Once `block_end` has reached the end of the container, this operator will do absolutely nothing.

The `operator*()` will return a `std::vector` containing copies of the elements in the current block spanned by `block_begin` and `block_end`. Note that this vector should be constructed each time `operator*()` is called.

The `operator==()` and `operator!=()` should compare the `block_begin` iterators. There is no need to compare the other two iterators.

Remember: All three of these templates have the same template parameters.

Requirement: It should only be possible to construct `Block_Iterator` objects through the `Block_Container` class, meaning `Block_Iterator` should have *no* public constructors.

There are testcases given in `given_files/program4.cc`, you are not allowed to modify the existing testcases. However you are encouraged to add more testcases.

5. In this assignment you will create a type trait called `largest_type`, which takes an *arbitrary* number of template parameters. This type trait must have a type alias called `type` which will be set to the largest (with respect to the `sizeof` operator) of all the passed in type parameters. [3p]

   You must also create a template alias called `largest_type_t` which acts as a short-hand for `largest_type<...>::type`.

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

   Hint: You can use `std::conditional` from the `<type_traits>` header and variadic recursion to implement this, but exactly how this is done is up to you.

6. **Discussion:** Explain why the compiler doesn't implicitly define the default constructor and the destructor for this struct: [3p]

```
struct My_Struct
{
  union
  {
    std::vector<char> v;
    std::string s;
    int i;
  };
};
```

How can we ensure that the *correct* member in the anonymous union gets destroyed when a `My_Struct` object falls out of scope? Discuss what we need to consider when assigning values to `v`, `s` or `i`.

7. **Discussion:** When talking about containers, one of the factors that needs to be considered is *iterator/pointer invalidation*. [3p]

   (a) What does *iterator/pointer invalidation* mean?

   (b) Choose three different containers and compare them with regards to iterator/pointer invalidation. What guarantees does your chosen containers have?

   (c) Give an example (either in code or textually) where iterator invalidation matters. You can speak in general terms, or you can give a specific example. Briefly explain *why* it matters in your chosen example.