LINKÖPING UNIVERSITY
Department of Computer and Information Science
Software and Systems
Christoffer Holm 2024-08-22

# Computer examination in
# **TDDD38** Advanced Programming in C++

**Date** 2024-08-22

**Time** 8-13

**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 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:* Unfortunately the cppreference search engine of choice, DuckDuckGo, seem to employ some sort of DOS protection, blocking our exam proxy after some time. The search functionality may work for some time if you do it through cppreference. You *cannot* search on DuckDuckGo.

1. For certain calculations it would be particularly useful to be able to represent and iterate all real values in an interval up to a specified numerical precision (i.e. up to a specified number of digits), for example: `0.01, 0.02, 0.03, ..., 0.99, 1.00` with two digits of precision. [5p]

   In this assignment you will implement a random access iterator (`https://en.cppreference.com/w/cpp/named_req/RandomAccessIterator`) called `Interval_Iterator` which takes a *non-type template parameter* called `precision` of type `std::size_t`. The template parameter `precision` represents how many digits of precision each number in the interval has. This iterator must fulfill all the operator requirements specified for a random access iterator. It must also provide the member types described here: `https://en.cppreference.com/w/cpp/iterator/iterator_traits`, you must determine what type each type alias refers to.

   **Example:** Suppose we have precision 3, and the interval $[0.015, 0.020]$ then our iteration will result in the following iteration order: `0.015, 0.016, 0.017, 0.018, 0.019, 0.020`. This means that the precision decides what the "next" number is, because if we instead had precision 4 we would get `0.0150, 0.0151, ..., 0.0198, 0.0199, 0.0200`.

   See `given_files/program1.cc` for testcases. Note that these are not necessarily enough so you should probably write some own testcases as well.

   **Requirement:** T iterator must store the current value as a `long long` rather than `double` to make sure that the accuracy is preserved. Suppose we have a precision of 4 and the current value in the iterator is 1.2647 then store the `long long` value 12647, and when dereferencing the iterator we divide 12647 with the factor $10^4$ i.e. 10000 to get 1.2647. To avoid repeated calculation you must calculate the factor *once*, *during compile-time*. This means you also have to implement either a `constexpr` function or a struct template that calculates $10^N$ for the positive integer $N$.

   **Note:** `double` accumulates rounding errors, so try to avoid working with `double` as much as possible in the assignment. Ideally it should only ever deal with `double` when returning from the dereference operator only.

   **Note:** This iterator allows for many interesting approximation techniques. For example, an iterable interval would allow us to calculate inverse values of *monotone* functions over a specified interval, i.e. inverses of functions that is either always increasing in value or always decreasing in value. If $f(x)$ is a monotone function (i.e. increasing or decreasing) over the interval $[a, b]$ then we can find an approximate solution $x$ to the equation $f(x) = y$, where $y$ is some specified value, by performing a binary search over $[a, b]$. See the final testcase in `given_files/program1.cc` to see how this technique together with the STL algoritm `std::lower_bound` (`https://en.cppreference.com/w/cpp/algorithm/lower_bound`) is used to solve the equation

   $$\sin(x) = 0.5$$

   where $x$ is somewhere between 0 and $\pi$.

   **Note:** The testcases uses an STL algorithm, but you don't have to use any in your solution. This assignment deals with iterators and templates, **NOT** STL.

2. **If you passed the midterm test then skip this assignment (you get full points).** [4p]

   The negation operator is well-defined for signed integers (invert the sign), bit sequences (flip each bit) and boolean values (apply the not-operator), but there are types where it is not well-defined at all. Examples of such types would be `std::string` or containers.

   In this assignment we will define negation of:

   - a `std::string` to mean reversing the order of the characters.
   - a container to mean recursively negating each of its elements.

   Your assignment is to implement a function template called `negate()` that takes an lvalue reference to a value of arbitrary type `T` and negates it according to the definitions specified above. Notice that what is meant by *negation* varies depending on properties of the type `T`. Specifically you need to handle four cases (in this order):

   1. If `T` is a `bool` then apply the *not*-operator (`operator!`).
   2. If `T` is a *signed* arithmetic type (e.g. `float`, `int`, etc.) then it should flip the sign of the value. **Hint:** `https://en.cppreference.com/w/cpp/types/is_signed`.
   3. If `T` is a container with `char` as its value type then it should reverse the elements in the container. **Hint:** `https://en.cppreference.com/w/cpp/types/is_same` and `https://en.cppreference.com/w/cpp/algorithm/reverse`.
   4. If `T` is a container of arbitrary value type, then it should iterate all of the elements and call `negate` on each of them. **Hint:** All containers have iterators.
   5. If none of the cases above apply, then do bitwise negation on the value, i.e. do `value = ~value` (note the "tilde" character, which is NOT a dash).

   **Requirement:** Several of these cases can potentially overlap with other cases, so it is important that a priority is induced. For example: `T = std::string` fulfills case #3 and case #4, while #5 overlaps with every case.

   There are testcases given in `given_files/program2.cc`, they all should work without modification.

   **Hint:** To implement the detection of the different cases, use SFINAE or concepts/requires-clauses for different overloads of a helper function. To induce priority you might want to add some additional parameters.

   **Requirement:** The `negate()` function must take its parameter by-reference and modify the passed in value. It should **NOT** create a new value. It should then return the parameter as a reference to make the function easier to test.

3. **Discussion:** What does the the `virtual` keyword do, and what reason do you see for C++ forcing you to explicitly mark functions as virtual instead of making all functions virtual by default? Make sure that you explain any costs that you mention. [2p]

4. **Discussion:** What different explicit type casts are there in C++? Describe at least three of them. Discuss what each cast does and how they differ from each other. Discuss why you think the casts where separated into different operations rather than just using C-style casts. [3p]

5. In modern computer science parallelism is of high importance. Being able to run multiple tasks at once is key for efficiency. Many times there will be bursts of tasks running and then some down-time where the CPU can power down. In `given_files/tasks.txt` there is a list of time intervals when tasks will be running. Your job is to find when bursts of tasks occur, i.e. longest intervals when CPU is *not* idle. [5p]

   **Requirement:** In this assignment you may not use loops, `std::for_each` or recursion. You must use STL algorithms and containers.

   In `given_files/program5.cc` there is type alias of `std::pair<int, int>` called `Task` as well as some relevant operations.

   In `given_files/program5.cc` there is a concrete example for how this is done, which you can look at while reading the following step-by-step instruction for how to implement this program:

   1. Read each task from `tasks.txt` into a vector of `Task` elements called `tasks`.

      **Hint:** Overload the stream operators for `Task` in the `std` namespace and then use the istream iterators.

   2. Sort all tasks using the default comparison operator.

   3. Once the tasks have been sorted we can build all partial bursts by partially summing over all tasks using the `merge_or_split()` function as the addition operator. This works as follows:

      (a) Pick the first task as the current burst.

      (b) If the next task overlaps the current burst, then extend the burst so that it covers that task.

      (c) If the next task does not overlap, then set the burst for the next iteration to exactly overlap the task.

      Since we are doing a partial sum, the result of each iteration will be saved, so we can later find all independent bursts of the CPU. Save all partial results directly in `tasks`.

   4. To make sure that we can find the bursts we sort the partial bursts based on their start points (in ascending order), and secondarily (if the start points are the same) based on the end points. This sorts the bursts according to their start points but ensuring that the first burst in each sequence is the longest one (i.e. the one that cover the most tasks).

   5. Remove all but the longest bursts that share the same start point (i.e. remove all duplicates when we **only** check the start point).

   6. Print all the remaining elements from the `tasks` vector.

6. **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]

7. In meta-template programming variadic templates are used as lists of datatypes. When doing things relating to this, it can be very helpful to implement operations that we commonly associate with lists of values. In this assignment you will implement a type trait called `Count_Unique` that takes a variadic template of arbitrary types and has a `static constexpr` variable called `value` that contains the number of *unique* types that occured in the passed in variadic pack. [4p]

   **Example:** The following expression `Count_Unique<int, float, float, int>::value` should return 2, one for `int` and one for `float`.

   In `given_files/program7.cc` there are a few testcases given as well as a helper struct called `Pack` that can be used to capture variadic packs as a single type.

   **Hint:** To implement this make a type trait called `Make_Unique` that takes a variadic pack and has a type alias called `type` that contains a `Pack` of all the unique types that occured in the passed in variadic pack. We can then, in the implementation of `Count_Unique` use the static member `size` inside the `Pack` that `Make_Unique` produces to get the number of unique types.

   The algorithm for checking unique elements in non-sorted lists can be implemented as follows (pseudocode):

```
List input { ... };

List make_unique(List result, size_t index)
{
  if (index >= input.size()) // if we've reached the end
    return result;
  else if (result.contains(input[index]))
    // this value has already been added to 'result'
    // so we just move on
    return make_unique(result, index + 1);
  else
  {
    // this is a new value, so we add it to 'result'
    // before we move on
    result.append(input[index]);
    return make_unique(result, index + 1);
  }
}
```

   To translate this into the type trait `Make_Unique`, take a normal type template parameter that is meant to contain a `Pack` of all the types that has currently been found. Notice that to extract the types from a `Pack` you have to make a specialization which matches `Pack<Ts...>` where `Ts` is a variadic pack.

   You will likely also have implement a way to find whether a `Pack` contains a particular type.

   **Note:** You don't have to do it as described in the hint if you find another way to do it.

   **Hint:** The type trait `std::conditional` can be useful (`https://en.cppreference.com/w/cpp/types/conditional`).