LINKÖPING UNIVERSITY
Department of Computer and Information Science
Software and Systems
Christoffer Holm 2023-01-12

Computer examination in
TDDD38 Advanced Programming in C++

Date 2023-01-12

Time 14-19

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. In the context of programming, a *macro* is similar to a variable, but instead of keeping track of a value it instead keeps track of a piece of text. Then, whenever a macro name is found in the code it will replace that name with the specified macro definition. [4p]

   This concept is however not exclusive to programming, it can be used in text processing to make short-hand for common terms or phrases. For example:

   Suppose we have to write ”Linköping University” a lot in our texts. Then we can define a macro, let's call it ”LiU” and let its definition be ”Linköping University”. Then we can write things like: `"I study at LiU"`, and it *expands* to (expands refer to the act of replacing the macro with its definition): `"I study at Linköping University"`

   One useful property that macros thave is that their definitions might refer to other macros. For example, suppose we have the following macros:

   ```
   ORGANIZATION = IDA / LiU
   IDA = Department of Computer and Information Science
   LiU = Linköping University
   ```

   Then the text: `"ORGANIZATION"` should expand to:
   `"Department of Computer and Information Science / Linköping University"`

   Meaning we had to recursively expand `IDA / LiU` inside the `ORGANIZATION` macro. This can occur in an arbitrary number of steps.

   In this assignment you will implement macros by following these steps:

   1. Implement a function called `define_macros` that takes an `std::ifstream&` called `ifs`. Each line in `ifs` has the following format:

      `<MACRO NAME>:<DEFINITION>`

      See `given_files/macros.txt` for examples.

      Use an appropriate container to associate a macro name with its definition. It should be possible to find a macro definition by just supplying its name. `define_macros` returns this container.
   2. Create a function called `expand` which takes a string containing a line and the macros container returned from calling `define_macros`.

      This function expands all macros in the line recursively and return the fully expanded line as a string.

      To implement this function you should go through the line word-by-word.

      Check if the word is a macro by looking for it in your container. If it is a macro, then call expand on its definition and put the result in the output string.

      If a word isn't a macro, then just add it to the output string (followed by a space).

      Note: It is important that we call `expand` on the macro definitions here rather than in `define_macros` since we don't know all the macros until after the `define_macro` is done.

   Requirement: You must use at least one *appropriate* STL algorithm and one *appropriate* STL container in your solution.

   There are example runs given in `given_files/program1.cc`. You should use `define_macros()` and `expand()` to implement a main program that takes an arbitrary number of lines (stop input with `<ctrl+D>`) and expand them using the macros defined in `given_files/macros.txt`.

2. **If you passed the midterm test, then you have full marks for this assignment.** [5p]

   In C++20 the popular library `libfmt` became integrated directly into the STL as the `std::format` function. This introduces a modern (and fast) way of dealing with string construction and formatting. In this assignment you will implement a very basic version of `std::format`.

   To do this you must implement two things:

   1. A class template called `Formatter` which is responsible for taking an arbitrary object of type `T` and printing it to a stream.
      This class implements a *static* function called `format` that takes an `std::ostream&` and a `T` object. The idea is that this function will print the passed in object to the stream.
   2. A function template called `format` that takes a formatting string and an arbitrary number of parameters of arbitrary types. The idea is that each occurance of `"{}"` in the formatting string will be replaced. The first occurance will be replaced with the first argument in the variadic pack, the second with the second and so on. Use `Formatter::format` to produce the correct format for each argument.

      Example: The call `format("{}:{}:{}", 1, 2, 3)` should produce the string `"1:2:3"`.

      Hint: To implement this you can use `std::ostringstream` and use variadic recursion to extract each argument.
      Hint: use the member functions of `std::string` to handle the formatting string.

   If `format()` finds a `"{}"` but there are no arguments left, then it should throw an appropriate exception. Likewise if `format()` has arguments left to handle but there are no `"{}"` left in the format string.

   You will also have to implement two *specializations* for the `Formatter` class:

   1. One specialization for `T = std::string` which formats the strings surrounded with the characters `<` and `>`.
   2. One specialization for `std::vector<T>` which recursively formats each element separated with spaces. The whole vector should also be surrounded with `{` and `}`.

   There are testcases given in `given_files/program2.cc`, these should **not** be modified. You can also check the testcases for more examples of how this should work.

3. **Discussion:** In general code it is quite common to vary the implementation of a function template based on the template arguments. There are two main techniques for doing this, *template specializations* and *function overloads*. Explain the difference between these two techniques. Are there circumstances when they are not interchangeble? Give at least one example where one of the techniques is better to use. Supply a short argument for why this is the prefered technique in that example. [2p]

4. The problem of checking whether a string matches a specific pattern is something that most programmers run into at some point in their career. A common solution is to use tools such as *regex* or *parser generators*. In this assignment you will construct a class hierarchy that represent various parsing operations that are used to solve this type of problem. [4p]

   The idea is that we create a base class called `Base_Parser` which contains the following virtual functions:

   - `parse()` which takes two *string iterators* called `begin` and `end`. The `begin` iterator should be taken as a reference.
     The purpose of this function is to check whether the position from `begin` to (at most) `end` matches a specific criteria. If it does, the `begin` iterator is updated to the position *after* the matching substring.
     `parse()` returns `true` if it could match the string, and `false` otherwise.
     If this function returns `false` then `begin` should be unchanged. If it returns `true` then `begin` should point to the first character that wasn't matched.
   - `copy()` which returns a *deep copy* of this parser.

   Note: `Base_Parser` does not represent any specific parser, so the virtual functions should **not** have any implementations.

   You must then implement (at least) three subclasses of `Base_Parser`:

   Match Stores a string called `match`. The `parse()` function of this class should check whether the first `match.size()` characters in the iterator range `[begin, end)` is equal to `match`. If there are fewer characters in `[begin, end)` than there are in `match` then it is **not** a match.
   Remember that `begin` should be unchanged if it didn't match.

   Alternate Contains a `std::vector` containing several arbitrary parsers (subclasses of `Base_Parser`). `Alternate::parse()` should try each parser in order until one of them succeeds. If no parsers succeed then the function returns `false`, otherwise it returns `true`.

   Sequence Also contains a vector of arbitrary parsers. `Sequence::parse()` should return `true` if all parsers succeed (in order). If any parser fails then the whole function should return `false` (and `begin` should be unchanged).

   Note: `Alternate` and `Sequence` have a lot of things in common, think about how you can reduce the code duplication by using inheritance and helper functions.

   Requirement: It must not be possible to get memory leaks in your solution.

   There is a *partial* testprogram given in `given_files/program4.cc`, you must modify it so that it works with your solution.

   Examples (bold and italic are user input)

```
$ ./a.out
(x)
Matched!
(x+y)
Matched
x*y
Didn't match!
```

5. **Discussion:** Below are two cases. For each case, identify a container that can be used to solve the problem *in-place* (i.e. by directly modifying the content of the container during traversals) *without moving the elements in memory.* The container **must** contain `int` values. Explain *why* this container works, and why another one doesn't. [4p]

   (a) While iterating the container, if we find the value `0` we want to insert another `0` at the beginning of the container, and insert `1` at the end.

   (b) While iterating the container, if the next element is equal to the current element then remove the next element.

6. In certain programming languages that doesn't support `class` or `struct` like things it is common to structure data with nested containers. Under these circumstances it can be useful to perform a so-called *flatten*-operation. [5p]

   A flatten-operation is when nested structures are turned into a single flat hierarchy. Meaning that all the elements stored deep within the nested structured are put into a single list.

   Example: Suppose we have the following structure:

```
{
  {
    { 1, 2, 3 }, { 4, 5 },
  },

  {
    { 6, 7, 8, 9 }, {  }
  }
}
```

   Then a *flatten*-operation results in: `{ 1, 2, 3, 4, 5, 6, 7, 8, 9 }`

   In this assignment you will implement a function template `flatten()` that takes two template parameters:

   - The type `T` which represents what data type the content of the flattened list should be (for example: in the example above `T = int`).
   - The type `Container` which represents a nested structure of containers (in the example above it would be something like: `std::vector<std::vector<std::vector<int>>>`).

   `flatten<T>()` should return a `std::vector<T>` containing the flattened result and it should take a `Container` parameter called `container`.

   In C++ we have one other way to store data indirectly, namely *pointers* and *iterators*, so our `flatten<T>()` function should cover three cases:

   1. If `Container` is a container you must recursively call `flatten<T>()` on each element. Merge all resulting vectors into one *flattened* vector which is then returned from the function.
   2. If `Container` is the same type as `T` then it will return a vector with only the value of `container` in it.
   3. If `Container` is a pointer *or* iterator (can easily be detected by checking if it can be *dereferenced*) then it should call `flatten<T>()` on the dereferenced value of the pointer/iterator and return the result.

   Note: There might be ambiguities between all of these cases. When that happens the *order* of the cases above determines the priority. Also note that if `T` is an iterator or pointer, then case #3 will not occur for that type.

   Note: There is **no need** to cover the case when `T` is a container. `std::string` are containers in the context of this assignment.

   There are testcases given in `given_files/program6.cc`.

7. **Discussion:** C++20 ranges are complementary to STL iterators and algorithms, but are *not* meant as a direct replacement. What are some differences between using range-algorithms and iterator-algorithms? Does range adaptors and views make a difference? [1p]