LINKÖPING UNIVERSITY
Department of Computer and Information Science
Software and Systems
Christoffer Holm 2022-06-02

Computer examination in
TDDD38 Advanced Programming in C++

Date 2022-06-02

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 this assignment you will implement something similar to a `std::tuple` by recursively nesting `std::pair`s together. Suppose that we want to create a list containing the values `1`, `2.3`, `'4'`. Then we can do this by using `std::pair` like this: [5p]

   ```
   make_pair(1, make_pair(2.3, make_pair('4', ???)));
   ```

   We will from now on call this type of object a *pair structured list*.

   One thing to remember with recursive structures is that we need a base case. For pair structured lists we need a way to mark the end of the list. To do this we simply create an empty class called `Sentinel`. Then we can create a pair structured list like this:

   ```
   make_pair(1, make_pair(2.3, make_pair('4', Sentinel{})));
   ```

   To make this easier to use you will have to create two function templates:

   - `make_list` which takes an arbitrary amount of parameters and constructs a pair structured list from the parameters.
   - `get_index` which takes an index `N` as a *template parameter*. It takes a pair structured list as a function parameter and then returns the `N`:th value of that list as a *reference*.

   There are a few testcases given in `given_files/program1.cc`. These should work without any modification.

   Hint: To implement `make_list` use variadic recursion. Make two overloads: one that takes no parameters and returns a `Sentinel` and another one that takes a variadic template and returns a pair where the first element is the first function parameter and the second element is the result of a recursive call to `make_list` with the rest of the function arguments.

   Hint: There are many ways to implement `get_index` and you are not limited to one way. As long as it works the intended way without modifying the given program.

   One suggestion however is to create a struct/class template `Get_Index` that takes an index `N` and an arbitrary type `T` as template parameters. `Get_Index` defines a static function called `value` that takes a reference to a pair structured list (which has type `T` here). Then you need to cover two cases:

   - For the primary template the static function `value()` should recursively call `value()` of the instantiation `Get_Index<N - 1, ???>` with the second value of the pair structured list as a parameter, and then return the result. This is the recursive step.
   - A specialization for when `N = 0` where the first value in the pair structured list is returned. This is the base case.

   You can then use `Get_Index::value()` to implement the function template `get_index()`.

2. **Discussion:** What is a dependent name? How is it related to the keyword `typename`? What ambiguity problems are related to dependent names? [2p]

3. Dynamically typed languages such as Python or Javascript can be (somewhat) simulated in C++ with for example `std::any`. But a big difference is that if we want to do something with `std::any` we have to convert them back to that specific type. In dynamically typed languages you don't have to know what the type of a value in order to use it. In this assignment we will make a variable type that approximates this behaviour in C++. [6p]

   To do this you will have to create a class `Variable` and a class hierarchy consisting of these classes:

   - `Variable_Base` a pure virtual (abstract) class that defines the interface for all possible values that can be stored in a variable. Has the following virtual functions:
     - `print()` that takes an `std::ostream` reference.
     - `add()` that takes an object from the `Variable_Base` hierarchy and returns a new `Variable_Base`. The idea behind this function is to determine what happens when we add two variables together. In this case it just throws an exception since the base class cannot exist.
   - `Number` is a subclass of `Variable_Base`. This class is the implementation of a variable that contains a number. It has a `double` data member called `value`.
     - `print()` simply prints `value` to the passed in `std::ostream`.
     - `add()` have the following behaviour: If the passed in parameter `rhs` is a `Number` then it creates a new `Number` which contains the sum of `this->value` and `rhs->value`. Otherwise it throws an exception.
   - `String` is a subclass of `Variable_Base`. This is the implementation of a variable containing a string. It has a string data member `value`.
     - `print()` simply prints `value` to the passed in `std::ostream`.
     - `add()` varies it behaviour depending on what the dynamic type of the passed in parameter `rhs` is: If `rhs` is a `Number` then it produces a new `String` which is the original string concatenated with the string representation of the passed in numbers value. If `rhs` is a `String` then a new `String` is returned which contains the original string concatenated with the passed in string. Otherwise it throws an exception.

   Using this class hierarchy we can then define `Variable` as follows:

   - It contains a data member `data` which can be any of the types in the `Variable_Base` hierarchy.
   - There are two constructors: one that initializes `data` as a `Number` and one that initializes it to a `String`. These constructors should take appropriate parameters.
   - An `operator+` that takes another `Variable` and calls `add` with the passed in variables `data` member as its parameter, and then returns a new `Variable` which contains the result of the `add` call. **Hint:** you might need to add a third constructor that takes a `Variable_Base` object.
   - Two `operator=`, one that takes a `double` and one that takes a `std::string`. These should replace the currently stored `data` with a new appropriate subclass initialized from the parameter.
   - Finally there is an overload for `operator<<` that calls the virtual `print` function of the stored `data` member in `Variable`.

   There is a given testprogram in `given_files/program3.cc`. This should **NOT** be modified.

4. In this assignment you will write a program that constructs a textual output based on a document template. In the context of this assignment we define a document template as a text file which contains the occasional string of the following format: `#N` where `N` is a positive integer. An example of such a file is given in `example.txt`. [5p]

   In this assignment you *must* use appropriate STL containers and algorithms. This means that you are not allowed to use any manual iteration statements (loops), nor recursion. You must also consider which container is best suited for each step in the program. It is possible to solve this assignment without `std::for_each` so refrain from using it.

   The program takes a document template file as its first command line argument. The rest of the command line arguments will specify (in order) what `#0`, `#1`, `#2` etc. will be replaced with. Most of the command line argument handling as well as the opening of the specified file is given in `given_files/program4.cc`. Your job is to implement the rest of the program, which is done by following these steps:

   1. Read each word of the file into a vector of strings called `text`.
   2. Create an associative container called `parameters`. For each element in the interval `argv+2` to `argv+argc` associate strings of the form `#N` (where `N` is an integer) to corresponding argument in the command line interval and store that association in `parameters` with `N` increasing by `1` each time (starting at `0`).

      Example: If the command line arguments contains the strings `"abc"` `"def"` and `"ghi"` then the association in `parameters` should be:

      ```
      "#0" -> "abc"
      "#1" -> "def"
      "#2" -> "ghi"
      ```

      And so on.
   3. In this step we must check that all strings of the form `#N` in `text` has an associated string in `parameters`. The program must report all *unique* strings of the form `#N` that has no association and then terminate the program. If all `#N` has an association, move on to the next step.

      Hint: Copy each string of the form `#N` in `text` that has no association in `parameters` to a separate container and then afterwards check whether the container is empty. If not, print all the strings in the separate container to `std::cerr` and then terminate. Remember that you may only print each unique `#N` once.
   4. Replace all occurances of `#N` strings in `text` with their associated strings in `parameters`.
   5. Print the content of `text` separated with a space to `std::cout`.

   Various example outputs are given in `given_files/program4.cc`.

5. **Discussion:** What is pointer (or iterator) invalidation in the context of containers? Give an example of when it can occur. Explain *why* this might cause issues. Are there any containers that are preferred if we want to avoid pointer invalidation? [2p]

6. Often we want to add elements to the beginning of a container, but not all containers support the function `push_front`. In those cases we might have to resort to using the more general `insert` function or even (in a few cases) `operator+`. [3p]

   In this assignment you will create a function template `prepend` that takes an arbitrary container and adds a value to the beginning of the container. This means that `prepend` takes two parameters: a container reference and a value.

   The implementation of `prepend` differs depending on certain properties of the passed in container. Specifically there are three cases which are prioritized in the order they are presented. All three cases will insert the value at the beginning of the container, but they will use different methods which are:

   - `push_front`
   - `operator+` (i.e. `container = value + container`)
   - `insert`

   You may **NOT** use C++20 concepts to solve this assignment.

   Requirement: This function should work for all containers that can be modified and where it is meaningful to "add a value to the front" (so associate containers, `std::array` and C-arrays should *not* be considered).

   Requirement: The value must be passed as a *forwarding reference* and must be properly forwarded to the functions/operators that are called in `prepend`.

   Hint: Create a general `prepend` function that calls a helper function which has one overload for each case. Note that you may have to add parameters to the helpers.

7. **Discussion:** What is the rule of zero and rule of five? How are they related? What special member functions are involved? In your opinion, what is the purpose of these rules? [2p]