LINKÖPING UNIVERSITY
Department of Computer and Information Science
Software and Systems
Christoffer Holm 2024-03-15

Computer examination in
TDDD38 Advanced Programming in C++

Date 2024-03-15

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:* The search functionality should work, but only if you do it through cppreference. You *cannot* search on DuckDuckGo.

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

   Reflection refers to features that allow programs to examine and modify its own behaviour. C++ does not have any reflection as a builtin feature (yet!), but it is possible to do some basic introspection (i.e. examination of the code itself) by using templates in a clever way.

   In this assignment you will create a template called `callable_info` that takes the data type of a *callable* object and extracts the return type and the type of the parameters as well as the number of parameters.

   A partial implementation of `callable_info` is given in `given_files/program1.cc`, but it is far from complete. However most of the extraction logic has been implemented. Recall that *callable* objects are either functions, function objects or lambdas (which are really just function objects in disguise).

   It is fairly straight forward to extract the parameter- and return types from a function or function pointer (see specializations #1 and #2 of `callable_info_helper`.

   To extract the parameter- and return type of a function object (and by extension lambdas) we must examine the member function `operator()`. Specializations #3 and #4 of `callable_info_helper` handles pointers to member functions.

   Note that `callable_info` is implemented as an alias template to the return type of the undefined `get_callable_info()` function.

   To finish the implementations, two things need to be implemented:

   - The type trait `extract_type` which takes an `std::size_t` called `N` and a variadic type template pack, and define a type alias called `type` that is the `N`:th type in the variadic pack.

     Note: `N = 0` should return the first type in the variadic pack.

     Note: The given helper struct `callable_info_base` uses `extract_type` in its implementation.

     Hint: Use variadic recursion, with `N` equal to `0` as the base case.

   - The function template `get_callable_info` which takes a template parameter `T`. This function should have two overloads:

     1. If `T` has the member function `operator()` then it should return `callable_info_helper` with its template parameter being the declared type of `&T::operator()` (note that you can check for existence of a member function by either trying to call it, *or* by checking that the expression `&T::operator()` is valid).
     2. Otherwise, return `callable_info_helper` with its template parameter simply being `T`.

     Note: All types will match the second case, so you must make sure that it is only ever picked if the first case is not applicable (i.e. utilize SFINAE and the overload resolution rules to induce a priority on the cases).

   It is probably easier to implement and test `extract_type` and `get_callable_info` in isolation before trying to integrate them with the given code.

2. **Discussion:** What is *placement new*? How is it different from `new`? When should it be used? [2p]

3. In `given_files/program3.cc` there is class `Sparse_Map` given which implements a very basic associative container called a *sparse map*. This container is generally optimized for fast iterations over all key-value pair with the cost being expensive insertion, deletion and lookup operations. *However* we can make all operations constant time if we use a lookup table for the expensive operations (more on this shortly). [4p]

   In this assignment you have three things you need to do:

   - Make the `Sparse_Map` into a template where `Key` and `Value` can be arbitrary types. You may assume certain properties of `Key` and `Value`: write what those assumptions are in a comment, if you don't you might get point deductions.
   - Make `insert()`, `erase()` and `at()` amortized constant-time with regards to `data.size()` by creating an appropriate lookup table that can be used to lookup what index a specific key is currently stored at in `data`. Note that this table must be an appropriate container that have amortized constant-time operations. It is also important to note that this lookup table should **NOT** be used during iteration, only during the specified functions: otherwise we are not utilizing the blinding speed of `std::vector` for our iterations. There are a few comments marked `TODO` in `given_files/program3.cc` which might help you realize how this can be achieved.
   - Implement an *bidirectional* iterator for this sparse map implementation. Read more here: `https://en.cppreference.com/w/cpp/named_req/BidirectionalIterator`
     Besides the normal requirements that a bidirectional iterator has, it must also fulfill the following requirements:
     - `operator*()` must return a pair containing a `const` reference to the key of the current element and a normal reference to corresponding value.
     - The iterator type must be public in `Sparse_Map`, but only `Sparse_Map` should be able to create iterators.
     - The iterator should have one data member, which is an iterator the `data` container in `Sparse_Map`.

   There are testcases given in `given_files/program3.cc`, you should test the `Sparse_Map` for different key and value types.

4. **Discussion:** Suppose we are iterating a container and whenever we find the value `0` we insert another `0` at the beginning of the container, and insert `1` at the end. Name *two* containers that can be used to do this *in-place* (i.e. by directly modifying the container rather than copying to a new one) *without moving the previously existing elements in memory.* [3p]

   Explain *why* your chosen containers work, and explain why `std::vector` does **not** work.

   It might be useful to look at this page: `https://en.cppreference.com/w/cpp/container`

5. In `given_files/processes.txt` there are several processes/programs listed, each with its own list of hardware resources that they use. At even intervals your operating system must decide which programs should run, but in order to do so it must take into account that certain resources might be busy. If a process depends on a resource that is busy, then that resource is not viable to run at this moment. [4p]

   In this assignment you will use STL algorithms and containers to determine which processes from `given_files/processes.txt` can run given a list of busy resources. There is a struct called `Process` given in `given_files/program5.cc`, you **must** use this struct in your solution. You **must** implement and use both stream operators for `Process`. You may not use any loops, recursion or `std::for_each` in this assignment.

   Your program may assume that a file called `processes.txt` exists, where each line contains a unique process on the following format:

   ```
   <name of process (may contain spaces)>: <list of resources separated by spaces>
   ```

   The user should then enter a list of busy resources to `std::cin` (when finnished entering busy resources the user presses `ctrl+D`). See examples in `given_files/program5.cc`.

   Note: You may assume that all lines in the file follow the specified format. I.e. no need for error handling.

   Hint: To read a line from a stream, use `std::getline()`. To process said line, then transfer it into a stringstream. Stream iterators can be used with any type that has corresponding stream operator.

   Hint: You can use `std::getline()` to read until a specific character (`:` in this case) from a stream.

   Hint: To find which processes can run, remove any processes that have any overlap whatso-ever with the list of busy resources (i.e. where the intersection of the process resource list and the busy resource list is non-empty).

6. The *Entity-Component-System (ECS)* pattern is a common way to design simulations and games. In this pattern we have *entities* that represent objects within the simulation/game, and each entity has a set of *components* which represents said entity having specific properties and/or data. Finally we have *systems* which processes components. A system only process those entities that have the specified component(s). There are many ways one can implement the ECS pattern. In this assignment you will create a *very* simple implementation of entities that stores arbitrary components. [5p]

   In `given_files/program6.cc` there is a given program that sets up a very simple simulation with 10 entities. There are three types of components: `Name`, `Health` and `Damage`. Each entity will be randomly assigned a set of components. Then we have two systems: one that processes all the `Name` components, and one system that processes all entities that have both a `Health` and a `Damage` component.

   To make this given program work, your job is to implement the `Entity` class, which must have the following member function templates (all of which takes a component type `T` as a template parameter):

   add takes a `T` object and stores that object in the entity.

   has returns `true` if the entity contains an object of type `T` and `false` if it does not.

   get gets a reference to the `T` object that is currently stored in the entity (you can assume that this function is only called when we *know* the entity has a component of type `T`).

   We need to keep track of which types are currently being stored in an entity, which can be done using `typeid` and `std::type_index`. See here: `https://en.cppreference.com/w/cpp/types/type_index`

   There are many ways to implement `Entity` and you are free to do it however you want as long as you don't modify the main program in any way. Below is a *suggestion* for how to do it using polymorphism. Create two classes:

   ComponentBase which stores a `std::type_index` data member that represents which type is currently stored in it. **Note:** this class must be polymorphic, so make sure there is at least one virtual function (for example the destructor).

   Component is a class template with type parameter `T` that inherits from `ComponentBase`. This class template should store a value of type `T`, and have a constructor that automatically sets the `std::type_index` stored in the base class to the `typeid` of `T`. It should also have a member function `get()` that returns a reference to the stored `T` value.

   Then entity should store a collection of `ComponentBase` pointers. When we add a component to an entity we simply allocate a `Component<T>` object (where `T` is the component type). To check if a component is a specific type, we simply examine the store `std::type_index` member.

   To actually retrieve a component of a specific type `T` in an entity, we loop until we find a `ComponentBase` pointer which stores that type (check the `std::type_index` member) and then cast that pointer to a `Component<T>` pointer and use the `Component<T>::get()` member function.

   Requirement: The given main program may not be modified in *any* way.

Requirement: There may not be any memory leaks and all destructors must run properly, and it should *not* be possible to copy an `Entity` object.

Note: There are a lot of aspects left up to you in this assignment, note that you will be assessed on how well you use the features available to you.

7. **Discussion:** Explain what an *xvalue* and an *lvalue* are. How are they different from each other? When do they appear? Give an example of an *lvalue*, an *xvalue* and a *prvalue*. [2p]