LINKÖPING UNIVERSITY
Department of Computer and Information Science
Software and Systems
Christoffer Holm 2020-01-13

Computer examination in

TDDD38 Advanced Programming in C++

Date 2020-01-13

Time 8-13

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 `given_files` (write protected). The exam will be available as a pdf in this directory at the start of the exam.
- Files for examination 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.

Available commands

`e++17` is used to compile with "all" warnings as *errors*.
`w++17` is used to compile with "all" warnings. **Recommended.**
`g++17` 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/` with the chromium browser, specifically you will *not* have access to anything in the language section of cppreference. You can start the browser by either running `chromium-browser` in the terminal or choose an appropriate option in the start menu. Do note that everything except cppreference will be unavailable. 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.

Theory questions

1. Answers may be given in either Swedish or English. Write all your answers in one text file and submit as assignment 1.

   (a) What problems are there in the following code? [1p]

   ```
   struct X
   {
     X(int x) : ptr{new int{x}}
     { }

     ~X() { delete ptr; }

     int* ptr;
   };

   int main()
   {
     X x{5};
     delete x.ptr;
   }
   ```

   (b) What is the difference between `if constexpr` and `if` statements? [1p]

   (c) Given: [1p]

   ```
   class A
   {
   private:
     class B { };
   public:
     B get() { return B{}; }
   };

   int main()
   {
     A a{};
     // create a variable of type B here
   }
   ```

   show how to create a variable of type `B` in `main` without *calling* `A::get` (i.e. `A::get` should not be evaulated).

   (d) Give an example of an *xvalue expression*. [1p]

   (e) What is a *union*? E.g. how is it different from `struct`? [1p]

Practical questions

2. Many terminal-based programs use what are known as Unix command line options. A good example of such a program would be `g++`. We specify which files to compile and which flags to use when calling the program, as such: `g++ -Wall program.cc`, which is a very typical usage of command-line options. [5p]

   In this assignment you are going to implement a very simple framework for dealing with such options. In this framework flags has to occur in a specific order. We will only deal with two types of options: *flags* (options that start with `-`) and *value arguments* (options that takes a value).

   In the example above `-Wall` is a flag and `program.cc` is a value argument that takes a string.

   This framework will be implemented with polymorphism. There is an example program given in `given_files/program2.cc`. This program implements some features of this framework, so your job is write the classes. Make sure to read the comments in the given file.

   You have to create these four classes:

   Option Base class for all types of options. Stores a name (as `std::string`) of the option that will be shown during error message. This name is set by passing its value to the constructor. `Option` has two virtual functions:

   `bool parse(std::string const& arg)` and `void print(std::ostream& os)`.

   `parse` will take a single argument from the command-line and return `true` if this option could parse the passed in argument, otherwise it returns `false`. In `Option`, `parse` doesn't have an implementation. `print` will print the name to `os`.

   Flag represents a flag option. This class stores a *reference* to a `bool` variable that indicates to the user that this flag was present. The constructor should set this variable to `false`.

   Inherits from `Option` and overrides `parse` and `print`. `parse` will set the `bool` variable to `true` and return `true` if the argument passed in is equal to the name of this option. Return `false` otherwise. The main program can then check the `bool` variable to see if this flag was present.

   `print` will print the name of this option inside brackets `[...]`.

   Argument represents a value argument. `Argument` is a class template with one parameter: `T` which represents the type of value it parses. This class stores a *reference* to a `T` variable called `target` where we will store that parsed value.

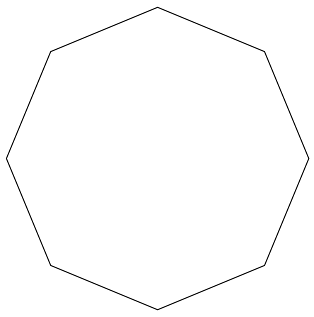
   Inherits from `Option` and overrides `parse`. `parse` will parse the passed in argument as a `T` value and set `target` to this value if the parse succeeded. Returns `true` if a value could be parsed, and `false` otherwise.

   Hint: Assume that `T` has `operator>>`, this is useful for converting a `string` to `T`.

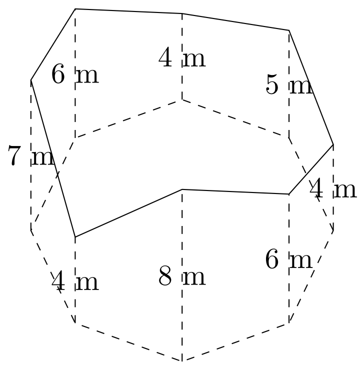
   Parser a class that holds a collection (`std::vector`) of options called `options`. Should have three functions: `add`, `parse` and `print` that should be possible to use as demonstrated in the main program. Implementations for `parse` and `print` are given in `given_files/program2.cc`. `add` should take an option and add it to the `options` collection.

   It should not be possible to copy any of the classes, and no memory leaks are allowed (cleanup should be done according to good C++ conventions).

3. There is a program given in `given_files/program3.cc` that calculates the length of a circular road, but it takes various height differences in to account as well. This circular road has the following shape when viewed from above: [5p]



Each line segment has the same length, stored as a constant called `length` in the program (expressed in meters). The road has a third dimensions as well: each vertex has its own height (expressed in meters) above sea level. See the following diagram:



The program represent this diagram by storing the height of each vertex as a `double` in the vector called `road`. The diagram above can be represented in the program with the values `4 5 4 6 7 4 8 6`.

The program will allow the user to enter heights (the number of heights will be the number of vertices). The program will then print the distance between each successive vertices as well as the total length.

Your assignment is to rewrite the code so that it uses only STL algorithms. The main goal is to make the code more readable by selecting algorithms that express your intent. A full solution should not require standard iteration statements, nor should it require `std::for_each`.

4. In C++20 a new concept will be introduced called *ranges*. A range is a pair of iterators, `first` and `last` that represents a range of data. The idea is to simplify code by replacing iterators with this concept. In this assignment you will implement a prototype of ranges. [5p]

   `range` is class template that takes one template parameter: `Container`, which represents the type of container this range spans.

   `range` must have two type-aliases (inner types):

   value_type represents the type of data that is stored in the underlying container.

   iterator represents the iterator type of the underlying container.

   `range` should have the following member functions:

   begin returns an `iterator` to the start of the range.

   end returns an `iterator` to the end of the range.

   size returns the number of elements in the range (the distance between the two iterators).

   take takes one parameter: `int n`. Will create a new range that spans the `n` first elements of the current range.

   skip takes one parameter: `int n`. Will create a new range where we skip the `n` first elements of the current range.

   Note: `take` and `skip` are not supposed to modify the underlying container, they should return a new range of iterators only.

   A range is easy to use, but only if we actually write algorithms for them. Therefore you should also implement three algorithms for the range. All of these algorithms take a reference to a range `r`. These algorithms are allowed to change the underlying container, but only through iterators.

   map takes a function object `op` and apply it to each element in the range. `op` takes one parameter of type `value_type` and have the same return type. `map` returns a reference to `r`.

   filter takes a function object `pred`. `pred` should take one parameter of type `value_type` and return `bool`. `filter` will move all elements for which `pred` returns `true` to the end of the range and then set the `last` iterator of `r` to the first of these moved elements (it should "remove" the values). `filter` returns a reference to `r`.

   reduce takes a `value_type` called `initial` and a function object `op`. `op` takes two parameters of type `value_type` and returns a `value_type`. `reduce` will combine all elements of `r` into one value with the help of `op` and `initial` (you can think of `reduce` as a general sum function). `reduce` returns the calculated value.

   Hint: try to use STL algorithms to implement these.

   It is very annoying to create a `range` as it stands right now since you have to specify the container-type and also pass it two iterators. Therefore you have to make a function `make_range` that takes a container and returns a range to that entire container.

   There are testcases given in `given_files/program4.cc`.

5. In this assignment you will create a function template `call_chain` that takes an arbitrary amout of callable objects (functions, function objects and lambdas) and returns a function object that will *chain* the passed in callables. [5p]

   What this means is that the following:

   ```
   auto f = call_chain(f1, f2, f3);
   cout << f(1, 2, 3) << endl;
   ```

   should be equivalent with:

   ```
   cout << f3(f2(f1(1, 2, 3))) << endl;
   ```

   There are more testcases given in `given_files/program5.cc`.

   To implement this, create a class template `call_chain_result` that takes an arbitrary amount of callable object types.

   `call_chain_result` have two partial specializations:

   1. A specialization that extracts the first template parameter in the given variadic pack as its own parameter. This one should store a data member `rest` that is a `call_chain_result` instantiated with all the callable object types, except the extracted parameter (variadic recursion).
   2. A specialization that only matches when there is only one template parameter.

   Both of these specializations have a data member `f` which is an instance of the first template parameter in the variadic pack.

   `call_chain_result` defines `operator()` taking an arbitrary amount of parameters as forwarding references. The behaviour of `operator()` varies between the two specializations:

   1. calls `f` with the passed in parameters and then passes the result to the callable object `rest`, returning the result.
   2. calls `f` with the passed in parameters and then returns the result.

   The function `call_chain` should return an instance of `call_chain_result`. This returned object should be callable with the same number of parameters as the first passed in callable object type and should have the same return type as the last callable object type. So for example, given:

   ```
   int first(int x, int y)
   {
     return x + y;
   }

   double last(int x)
   {
     return 0.5 * x;
   }
   ```

   The following:

   ```
   auto fun = call_chain(first, last);
   ```

   should result in a variable `fun` that can be called with two `int` parameters and returning a `double`. I.e. `fun(1, 2)` should return the `double` value `1.5`.