LINKÖPING UNIVERSITY
Department of Computer and Information Science
Software and Systems
Christoffer Holm 2018-08-22

---

Computer examination in

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

---

**Date** 2018-08-22

**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.

## 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) Why won't the program compile if `A::operator delete` is a private member of `A` in the following example? [1p]

   ```
   int main()
   {
     A* a = new A{};
     // explicitly not deleting 'a'
   }
   ```

   (b) What is *copy elision*? [1p]

   (c) How does *ADL* (argument dependent lookup) come into play here? [1p]

   ```
   #include <iostream>
   int main()
   {
     std::cout << "Hello World!" << std::endl;
   }
   ```

   (d) Give an example of a *scoped enum* declaration. [1p]

   (e) What does it mean if a function has the *strong exception guarantee*? [1p]

## Practical questions

2. You are about to host a dinner party for your friends. You have a selection of various foods such as pizzas, pizza rolls and salads. As with all people, your friends have various preferences when it comes to food and being a good host you want to accomodate all of your friends if possible. You have four types of friends; those who eat everything, those who love pizza but dislike pizza rolls, those who love salad, and those who are vegetarians. [5p]

   Your assignment is to create class hierarchies of the various foods and guests at your dinner party. The following classes should be implemented:

   **Food** abstract base class of all food items. This class contains a name and a pure virtual function `is_vegetarian`.

   **Pizza** dervied class of `Food` which stores a `bool` that determines whether or not this is a vegetarian pizza. This class must override `is_vegetarian` such that it will return the `bool` variables value.

   **Pizza_Roll** is exactly the same as `Pizza` with the exception that this class appends the word "roll" after the name.

   **Salad** Derived class of `Food` which overrides `is_vegetarian` such that it always returns `true`.

   **Guest** base class of all guest types. Contains the name of the guest, a virtual function `prefer` which takes an object of type `Food` and returns `true` and a function `eat` which takes a `Food` object and prints a message whether or not the guest prefers that food. The message should have the following format: `<name of guest> eat <name of food>.` if the guest prefers the food and `<name of guest> does not want <name of food>.` if the guest does not prefer the food.

   **Salad_Lover** derived class of `Guest` which overrides `prefer` such that it returns `true` only when the supplied `Food`-parameter is of type `Salad`.

   **Pizza_Lover** derived class of `Guest` which overrides `prefer` such that it returns `true` if the `Food`-parameter is *exactly* of type `Pizza`, otherwise it should return `false`.

   **Vegetarian** derived class of `Guest` which overrides `prefer` such that it returns `true` only when `is_vegetarian()` is `true` for the given `Food`-parameter.

   There is a skeleton for a test program given in `given_files/program2.cc`.

3. Iterators have quite a wide range of applications. Iterators can be used to iterate over containers (the usual case), but they can also be used to perform some kind of operations over containers, one element at a time. [5p]

   In this assignment you are to implement a class called `fold` which stores a container, a binary operator and some initial value. `fold` will then provide an iterator range with iterators of type `fold_iterator` through the member functions `begin()` and `end()`. Dereferencing `fold_iterator` will return the sum of all elements from the beginning of the range up to and including the current element that the iterator points to. This sum should by default be calculated using `operator+`, but the user should be able to specify their own binary operator to be used instead.

   The sum will start at some initial value which can be specified by the user and should have a default value (should correspond to a default initialized element in the container).

   `fold` should have two template-parameters; the type of a container and the type of a binary operator. The binary operator should have the default type `std::plus<T>` where `T` is the type of elements in the container.

   The constructor of `fold` should take three parameters; a container reference, an initial value and an instance of a binary operator function object. Both the initial value and the binary operator should have default values, i.e. the constructor should be possible to call with either 1, 2 or 3 parameters.

   `fold` should contain two functions `begin()` and `end()` which returns `fold_iterators` corresponding to the begin and end of the container.

   `fold_iterator` must be a *ForwardIterator*, with both post- and prefix increment operators, `==`, `!=` and a derefence operator which returns the value type of the iterator (should correspond to the type of the elements in the original container). Note that `operator->` is not required in this assignment.

   If everything is implemented correctly, then the following code should output `1 3 6 10`.

```
vector<int> container{1,2,3,4};
for (auto p : fold{container}) {
  cout << p << " ";
}
```

   A test program is given in `given_files/program3.cc`.

4. In this assignment, it's extra important for you to use the standard library in a good way. To get full marks, use the correct algorithm for the given task – `std::for_each` is not a valid solution for all tasks and you won't need any handwritten loops. [5p]

   The given file has a simple function to test if a value is a prime number and a simple type alias called `num_type`.

   Your task is to create a program that generates a random number with its prime factors by using following the algorithm below.

   1. Create a `vector<num_type>` with space for 10 elements
   2. Fill the vector with random values in the range $[2, 75]$
   3. Remove (with the help of the given `is_prime` function) the numbers that aren't prime numbers
   4. Sort the remaining values
   5. Print the remaining numbers to standard output
   6. Calculate the product of the remaining numbers
   7. Print the product to standard output

   A function `is_prime()` that checks if a number is prime is given in `given_files/program4.cc`.

5. In this assignment you are to implement a template function `prompt()` which takes an arbitrary type `T` as template-parameter and an `std::istream&` as function-parameter. [5p]

   The template function `prompt()` should read a value of type `T` from the supplied `std::istream` and then return said value.

   The behaviour of `prompt()` should change if `T` is a:

   **std::string** then it should read the entire line (using `std::getline()`) and return the read string;

   **Container** then it should read individual values from the stream and push them to the back of the container. These values should be read as long as it is possible i.e. something along the lines of:

   ```
   while (is >> value)
     container.push_back(value);
   ```

   Partial credits will be given if only the `T` and `std::string` specializations are implemented.

   There are some testcases and a general implementation of `prompt()` in `given_files/program5.cc`.

   **Hint:** Create a template class/function `prompt_helper` and use this one to create all the possible specializations. If you are having trouble that the compiler chooses the wrong overload, try to add an extra argument to `prompt_helper`.