

---

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

---

**Date** 2024-01-11

**Administrator**

**Time** 8-13

Anna Grabska Eklund, 28 2362

**Department** IDA

**Course code** TDDD38

**Teacher on call**

**Exam code** DAT2

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.

**Examiner**

Klas Arvidsson (klas.arvidsson@liu.se)

### 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 handed 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. Iterators and ranges are a very useful construction that generalizes iteration to a broad degree. However they are all based around iterating ranges *in order*. In this assignment you will create a construction that allows us to iterate things out of order, something like this:

[5p]

```
std::vector<int> v { 2, 3, 5, 7 };
for (int n : select_indices(v, { 0, 0, 3, 1, 2 }))
    std::cout << n << " ";
```

Where the `select_indices()` function determines based on *indices* what order the elements in `v` should be accessed in. Here the resulting output should be: 2 2 7 3 5 since the passed in indices were { 0, 0, 3, 1, 2 }.

To do this you must create two class templates:

**Index\_Iterator** which takes one type template parameter called `Value_It`. It has two data members: `index` which has type `std::vector<std::size_t>::const_iterator` and `value` which has data type `Value_It`. Note that `index` is the current index we want to access in the range starting at the `value` iterator.

The **Index\_Iterator** class template is a forward iterator, so it must fulfill the following requirements:

- It must have type aliases `value_type`, `reference`, `pointer` and `difference_type`. All of these are set to corresponding aliases from `Value_It`. Other than these it must also have the type alias `iterator_category` which is set to the type `std::forward_iterator_tag`.
- It must have an appropriate dereference operator (`operator*`). This operator accesses the current index (retrieved by dereferencing `index`) at the range starting at `value`. **Note:** `value` should never be updated, it should always refer to the start of a range. Instead you have to construct a new iterator (by using `std::next`) and dereferencing that.
- Both the prefix and postfix increment operator (`operator++`). These should increment the `index` iterator but **NOT** the `value` iterator.
- Comparison operators (`operator==` and `operator!=`) that simply compares all iterators.

The constructor of **Index\_Iterator** should only be callable by the **Index\_Range** class template (this is done by making the constructor private and then marking **Index\_Range** as a friend).

**Index\_Range** which takes one type template parameter called `Container`. It has two data members, a *reference* to a `Container` object and an `std::vector<std::size_t>` member called `indices`.

**Index\_Range** contains two member functions: `begin()` and `end()` both of which return corresponding **Index\_Iterator** objects.

The constructor of **Index\_Range** takes a `Container` reference, which is used to initialize corresponding data member, and an `std::initializer_list<std::size_t>` which is then used to initialize the `indices` vector.

You must also create an appropriate `select_indices()` function template that takes an arbitrary container and an `std::initializer_list<std::size_t>` and returns an appropriate **Index\_Range** object. There are testcases given in `given_files/program1.cc`.

**2. If you passed the midterm test then skip this assignment (you get full points).**

[5p]

In template meta programming we usually think of variadic packs as lists of data types. In this assignment you will implement a few type traits that perform queries on variadic packs.

In `given_files/program2.cc` there is a struct template called `Pack` which is used to represent a list of data types (note that it is empty since this is just used to bundle variadic packs together).

You must create three type traits:

**Contains** which takes two template parameters: an arbitrary type `T` and a `Pack` containing an arbitrary number of types.

This type trait contains a `constexpr static bool` variable called `value`. This variable should be `true` if *any* of the types in the `Pack` is the same type as `T`, and `false` otherwise.

This type trait is undefined if the second template parameter is *not* a `Pack`.

**Is\_Subset** takes two template parameters: specifically two `Pack` types.

This type trait contains a `constexpr static bool` variable called `value`. This variable should be `true` if *all* types in the first `Pack` are present in the second `Pack`, `false` otherwise.

**Note:** You can go through each type in the first `Pack` and use `Contains` to check whether each type is contained in the second `Pack`.

**Are\_Equal** takes two template parameters: specifically two `Pack` types.

This type trait contains a `constexpr static bool` variable called `value`. This variable is `true` if both `Pack` types are subsets of each other, otherwise `false`.

**Note:** You can use `Is_Subset` twice to implement this.

In `given_files/program2.cc` there are a few testcases given.

**Hint:** To make sure that your parameters are `Pack` types, and to have direct access to the variadic packs, you can create empty primary templates and then use template specialization to specialize for the case when the parameters are `Pack` types containing a variadic pack.

**Hint:** You can solve this using either variadic recursion or using fold expressions. It might be helpful to use the `std::is_same_v` type trait defined in `<type_traits>`.

3. **Discussion:** What is an `enum`? What is the difference between a *scoped enum* and *unscoped enum*? Give an example of when an `enum` could be appropriate to use. [2p]

4. `std::variant` is a very useful and modern way to represent sum types. A common usecase for it is to have functions that return values of one of two types, which it does well. However `std::variant` is built for having *many* alternatives. In this assignment you will build a simpler version of `std::variant` that only has two possible data types. [4p]

Create a class template called `Either` which takes two template parameters `T1` and `T2`. It contains an anonymous union which has a field of type `T1` called `first` and another field of type `T2` called `second`. `Either` also contains a bool called `is_first` which is used to keep track of which of the two fields in the anonymous union that is currently active (if `true` then `first` is active, and if `false` then `second` is active).

`Either` has two constructors, one that initializes `first` and one that initializes `second`. Remember to appropriately set `is_first` as well.

You must define a destructor which destroys the *appropriate* field, i.e. if `is_first` is `true` then it calls the destructor of `first`, otherwise it calls the destructor of `second`.

There should be two assignment operators, one for when the user assigns a `T1` and one for when the user assigns a `T2`.

**Note:** remember that when it is changing active field you **must** use *placement new* to construct a new object in the inactive field (remember to call the destructor of the previous field as well).

`Either` has three other member functions:

- `has_first()` which returns the value of `is_first`.
- `get_first()` which returns the first field as a reference *if* `first` is the active field, otherwise it should throw an `Either_Error`.
- `get_second()` which returns the second field as a reference *if* `second` is the active field, otherwise it should throw an `Either_Error`.

**Requirement:** there should be no memory leaks, make sure to check with `valgrind`.

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

5. **Discussion:** What is the difference between using parentheses when initializing a variable, and using curly braces? Give a list of what steps each initialization tries. Are there any other differences that are not covered by the steps? [2p]

6. **Discussion:** Which are the special member functions? Explain the purpose of each of them and include example code where they might be called assuming the compiler performs no optimization. Your explanation should also include how the special member functions are declared. When is the auto generated implementations enough and when should you implement your own versions? [3p]



7. From mathematics we are familiar with polynomials, i.e. functions that have the form:

[4p]

$$p(x) = c_0 + c_1x + c_2x^2 + \dots + c_nx^n$$

Where  $c_0, c_1, c_2, \dots, c_n$  are the so-called *coefficients* and  $x$  is a variable. When *evaluating* a polynomial this means replacing  $x$  with a specific value and then calculating the result of the expression.

In this assignment you will use STL algorithms to create a string representation of a polynomial *and* to evaluate said polynomial.

The program should read all coefficients from a *single* line of input from `std::cin` (the coefficients are of the type `double`). The coefficients are separated with a space. The first coefficient the user enters should be  $c_0$ , then  $c_1$  and so on. Once the user is happy with their coefficients they press `<Enter>`. Note that the number of terms in the polynomial is determined by the number of coefficients the user enters.

**Hint:** use `std::getline()` and `stringstreams` to achieve this. You will do this again later in the assignment so you should put the reading logic in a separate function.

Once the user has entered the coefficient, you should print the polynomial according to the execution examples in `given_files/program7.cc`. How you do this is up to you, but remember to use STL algorithms and appropriate containers.

Finally the user should be prompted for a list of values on a *single* line (exactly the same way we read the coefficients). These values are also `double` values. Then for each value you should evaluate the polynomial and print the result according to the format seen in `given_files/program7.cc`.

**Hint:** Evaluating the polynomial can be done by creating a secondary vector called `values` which contains  $x^0, x^1, x^2, \dots, x^n$ , where  $x$  is the value we want to evaluate and then take the inner product of this list with the coefficients.

**Hint:** Use `std::pow` from `<cmath>` to calculate powers.

**Requirement:** You may not use any loops, recursion or `std::for_each`.