ÖZYEĞİN UNIVERSITY
Faculty of Engineering
Department of Computer Science

CS 409/509
Advanced C++ Programming
Midterm Project 1

Duration: 3 days
Exam Starts at 21:00, May 4th, 2021
Upload solution to the LMS by 21:00, May 7th sharp.
Uploading within 21:00-22:00 range will result with 20% penalty.

General Instructions:

- This is an **open-book** project/exam. You **MAY** use any other supplementary material.
- **You SHALL NOT ask anything to another person during the exam (I.e. asking questions in stackoverflow.com or similar sites are forbidden).**
- Read the questions **carefully**; make sure that you understand the problems well before solving them.
- Write tidy code.
- **You will upload single file with name "midterm1.cpp" that compiles with latest versions of either Clang, GCC, or MSVC in C++20 standards mode. First line of your file must contain your name and your student id respectively as a code comment.**

Grading:

1a (10)	**1b** (5)	**1c** (5)	**2** (15)	**3a** (10)	**3b** (10)	**4** (10)	**5** (5)	**6** (15)	**7** (15)	**8** (10)	**TOTAL** (100)

You are given a code snippet below.
Questions are written in black bold in the below code. Only fill in the spaces that are indicated in the global namespace, do not touch any other code block. Doing so will make you lose points.
Console outputs that you are required to write your code to support them exactly are written in brown bold.
Copy the code below to your editor and create "midterm1.cpp" file and start solving it.

```
// ...
// Fill in all necessary extra code in this space to satisfy all questions below.
// You cannot change any other content other than the places marked as red.
// ...

// StudentID: xxxxxxxx, Name: AAA BBB CCC
#include <iostream>
#include <tuple>
#include <vector>
#include <string>
#include <map>
#include <variant>

using namespace std;

template<typename ...> struct TD;

// Solve Q1a (10pts), Q1b (5pts) here:
// LIMITATION! For Q1a and Q1b in total, you can use at most 6 semicolon (;) characters
including the struct ending semicolon


// Solve Q1c here (5pts)

// Solve Q2 here (15 pts)

// Solve Q3a (10pts) and Q3b (10pts) here

// Solve Q4 here (10 pts)

// Solve Q5 here (5 pts)

// Solve Q6 here (15 pts)

// Solve Q7 here (15pts)
// A String class to hide the underlying details of std::string (i.e. std::basic_string<char,
...>) This shows as "String" when type-debugging, and facilitates
struct String : string { using string::string; };

// The transformer required by Q7 is supplied here
template<typename ...>  struct Transformer;
template<typename T>    struct Transformer<T>       { using type = T; };
template<>              struct Transformer<char>    { using type = int; };
template<>              struct Transformer<short>   { using type = int; };
template<>              struct Transformer<long>    { using type = int; };
template<>              struct Transformer<double>  { using type = float; };
template<>              struct Transformer<string>  { using type = String; };


// Solve Q8 here (10 pts)

int main()
{
    // LIMITATION! For Q1a and Q1b in total, you can use at most 6 semicolon (;) characters
    // including the struct ending semicolon. More than 6 semicolons, you get zero points.
    // Q1a - Create a Mat<T> class with a constructor of Mat(rows, cols, initial_value)
    // T is automatically deduced from the initial_value's type
    auto m1 = Mat(2, 3, 9.9); // 2 row, 3 column matrix with double values is initialized to 9.9
for each cell
    print(m1);

    // Q1b - Make below assignment style work (i.e. m1[rowindex][colindex])
    for(size_t i=0; i<m1.rows; ++i)
        m1[i][i] = 1.1;
```

```
    // Q1c - write a print free-function that prints an instance of Mat<T>
    // if T is string, it puts " around the value, otherwise it directly prints the value
    print(m1);

    // Q2 - Write a "auto transform(const Mat<T>& mat_src, auto&& func)" free-function
    // that can take a source matrix and transform its contents globally.
    // At the end it returns the transformed matrix. Original matrix stays untouched.
    // Returned matrix can be of a different type depending on the "func"'s return value.
    // i.e. Mat<int> after transformation can be Mat<double> for instance.
    auto m2 = transform(
        Mat{2, 1,
            map<string, variant<string, double, int>>{
                {"pi", 3.14},
                {"CS", "409/509"},
                {"year", 2021}
            }
        }, []<typename T>(T&& map_) {
            auto s = string{};
            for(const auto& [key, value] : map_)
            {
                auto value_str = string{};
                if(holds_alternative<string>(value))
                    value_str = get<string>(value);
                else if(holds_alternative<double>(value))
                    value_str = to_string(get<double>(value));
                else if(holds_alternative<int>(value))
                    value_str = to_string(get<int>(value));
                s += key + ": " + value_str + " ";
            }
            return s;
        });
    print(m1);
    print(m2);

    // Q3a - Write a SINGLE (overloading inc is not allowed) free-function named "inc".
    // writing overloads for inc function will get you zero points
    // This function returns a new Mat<T> whose contents are incremented by 1.
    // It also writes l-value or r-value to the console based on its parameter's situation at
the call site.

    // Q3b - Make the SINGLE "inc" function available only for Mat<T> types using concepts
    // writing overloads for "inc" function will get you zero points
    // For instance, Mat<string> cannot be incremented. Mat<int>, Mat<float>, ... can be
incremented.
    print(inc(m1));
    print(inc(Mat(1, 4, 1)));


    // Q4 - Write a concat struct that can concatenate tuples at least as described below
    // concat omits the void at the end
    // concat concatenates types of two tuples into one tuple
//    using T1 = tuple<int, double, float>;
//    TD< concat_t<T1, void> > q4a; // ---> tuple<int, double, float>
//    TD< concat_t<T1, T1> > q4b; // ---> tuple<int, double, float, int, double, float>


    // Q5 - Write IsIntegral value-trait which is similar to std::is_integral.
    // But your implementation must also accept IsIntegral<> as a valid entry.
    // i.e. <> means an empty parameter-pack
```

```
    // Q6 - Write "filter_types" type-trait
    //  that accepts a value-trait and many types
    //  as a value trait you must support at least both of IsIntegral<> and is_integral<void>
    //  a value-trait can be, for instance, IsIntegral that checks if a type is suitable or not
    //  in the end, filter_types struct supplies the types filtered according to the value-trait
in its "type" attribute
    //  Do not write templated classes in the main() function block. Leave them in the global
namespace.
    using TUPLE = tuple<int, float, string, char, short, double, string, double, float>;

    using TUPLE_INTEGRAL = filter_types_t<IsIntegral<>, TUPLE>;
//    TUPLE_INTEGRAL --> tuple<int, char, short>
//    TD< TUPLE_INTEGRAL > q6a;

    using TUPLE_FLOATING = filter_types_t<is_floating_point<void>, TUPLE>;
//    TUPLE_FLOATING --> tuple<float, double, double, float>
//    TD< TUPLE_FLOATING > q6b;

    // Q7 - Write a "transform_types" type trait
    //  that accepts convertion type-trait and many types
    //  a type-trait specialized for your scenario is already supplied. Its called
"Transformer".
    //  in the end, transform_types struct supplies the transformed types according to the type-
trait in its "type" attribute
    //  Do not write templated classes in the main() function block. Leave them in the global
namespace.

    using TUPLE_TRANSFORMED = transform_types_t<Transformer<>, TUPLE>;
//    TUPLE_TRANSFORMED --> tuple<int, float, String, int, int, float, String, float, float>
//    TD< TUPLE_TRANSFORMED > q7a;

    // Q8 - Write a constexpr free-function named count_types
    // when used as shown below it returns the number of types in a tuple that fits to the
criterion supplied
    cout << "Number of integral types in TUPLE is " << count_types<IsIntegral>(TUPLE{}) << endl;
    cout << "Number of integral types in TUPLE is " << count_types<is_integral>(TUPLE{}) <<
endl;
    cout << "Number of floating types in TUPLE is " << count_types<is_floating_point>(TUPLE{})
<< endl;

    return 0;
}
```

OUTPUT

```
9.9 9.9 9.9
9.9 9.9 9.9

1.1 9.9 9.9
9.9 1.1 9.9

Transforming r-value parameter
1.1 9.9 9.9
9.9 1.1 9.9

"CS: 409/509 pi: 3.140000 year: 2021 "
"CS: 409/509 pi: 3.140000 year: 2021 "

Incrementing l-value matrix
Transforming l-value parameter
2.1 10.9 10.9
10.9 2.1 10.9

Incrementing r-value matrix
Transforming l-value parameter
2 2 2 2

Number of integral types in TUPLE is 3
Number of integral types in TUPLE is 3
Number of floating types in TUPLE is 4
```