CSCI 235 Exam Rules

- By taking part in the exam, you confirm that you have read the exam rules, and agree with them.
- Students are not allowed to look at the questions, until permission is given by the proctors.
- Write your name (first imya, then familya) on every sheet of paper that you wish to be checked.
- Answers must be written in readable fashion in blue or black color. Use of red color is not allowed, because it is used for grading.
- Answers must be well-structured, and unambiguous. Graders may interpret ambiguous answers in every way that they choose.

  Present your answers in such a way that the graders don't have to search for them.
- Don't write very small. Graders are not obliged to read very bad handwriting.
- It is not allowed to use of any form of electronical equipment during the exam. (mobile phones, computers, etc.). It is also not allowed to use any notes during the exam.
- Students may not take out any written materials from the room.
- Students may request for permission to leave the room shortly. Such a permission need not be granted. Leaving without permission or longer than agreed, means not coming back before the exam is over.
- If you cheat, you are taking away value from the grades that were obtained in honest fashion. You are not harming your teachers, but your fellow students who didn't cheat. Cheating may result in zero points, and a report to the Dean of SST
- Don't come begging for a better grade after the exam. It is allowed to have a look at the exam with the purpose of improving yourself. It is also reasonable to expect that true grading errors are corrected, but it is not OK to ask for better grade if you think that the exam was checked too strictly. We apply the same rules to everyone equally.

  If you request one part of your work to be regraded we may decide to recheck your complete exam.

CSCI 235, Programming Languages, Midterm Exam, C^{++}

Time: Thursday, 04.10.2018, 12.00-13.30

1. What is inlining? Mention two advantages of inlining. Where should the **inline** keyword be placed? Where should inlined functions be defined?
2. What are the time complexities of the following operations? (use O notation)
   (a) Insertion at an arbitrary place in `std::list`.
   (b) Deletion at arbitrary place in an `std::vector`.
   (c) Lookup of a key in `std::map`.
   (d) Deletion of a key/value pair in `std::unordered_map`, assuming that the key is given.
3. Consider the following C^{++} files (a chess program):

chess.cpp

```
#include <iostream>
#include <unordered_map>
    // for storing reachable states

#include "board.h"
#include "move.h"
#include "piece.h"
```

board.cpp

```
#include "board.h"
```

board.h

```
#include <vector>
#include <iostream>
#include "piece.h"
```

piece.cpp

```
#include "piece.h"
```

piece.h

(a) Assuming that the executable will be called **chess**, write the **Makefile** for this project. Assume we are using `c++-14`. (There are always

g++ -o ~.o ~c++-14

chess.O: chess.O board.O move.O piece.O

a few students, who will view this as an occasion to show how to smart they are. This is not such an occasion. Follow the pattern that was used in the lecture.)

(b) Write the include guard for file **piece.h**. What is the purpose of an include guard?

4. We have seen the datatype `std::string` in the STL, and we have seen own string implementation.

(a) Sort the following strings with the alphabetic, lexicographic order:

ondassyn, banu, aba, aabc, dimension, bank, on, dim,
ban, z, aa, zz, dinmukhamed, nazgul, hans.

(b) In order to be usable with `std::map`, which properties must a binary relation have?

(c) We will implement strings as list of characters, i.e. as `std::list<char>`. For example, we have

```
std::list<char> good = {'g','o','o','d','m','o','r','n','i','n','g'};
```

Write the following operators:

```
bool operator < ( const std::list<char> & l1,
                  const std::list<char> & l2 );
   // Implements alphabetic, lexicographic order.
bool operator == ( const std::list<char> & l1,
                   const std::list<char> & l2 );
   // Implements equality.
```

(d) Define the remaining four comparison operators, using short **inline** functions.

(e) Write

```
const std::list<char> &
operator+= ( std::list<char> & s1, const std::list<char> & s2 );
```

The operator must give correct results in the case of self assignment.

(f) Also write

```
std::list<char> operator + ( std::list<char> s1,
                             const std::list<char> & s2 );
```

The implementation must be short, use `operator +=`, and move the result.

(g) Finally, write

```
std::ostream& operator << ( std::ostream& out,
                            const std::list<char> & s )
```

that prints `s` as a string. The function must use **range-for**.

5. Consider the implementation of **tree** that is attached to the exam.

   (a) Write the copy constructor, the moving copy constructor, the copying assignment, and the moving assignment for **tree**.

   (b) Write the destructor for **tree**. Note that you have to write a helper function `destroy( tree_node* n )`.

   (c) Add a method `size_t size( ) const` to **tree**. This method also needs a helper function.

   (d) Method `find` does not behave in the same way as `operator[]( )` and `at( )` on `std::map` and `std::unordered_map`. Describe the differences.

   (e) Write `std::string& operator[]( const std::string& )`. It must behave analogously to `operator[]( )` on `std::map` and `std::unordered_map`.

```
#include <string>
#include <iostream>
#include <list>



struct tree_node
{
   std::string key;
   std::string value;

   tree_node* left;       // Both can be nullptr.
   tree_node* right;

   tree_node( const std::string& key, const std::string& value,
              tree_node* left, tree_node* right )
      : key{ key }, value{ value },
        left{ left }, right{ right }
   { }

};



tree_node* copy( const tree_node* n )
{
   return new tree_node( n -> key, n -> value,
      ( n -> left ) ? copy( n -> left ) : nullptr,
      ( n -> right ) ? copy( n -> right ) : nullptr );
}



// You don't have to care about balancing the tree.

void insert( tree_node* & n,
             const std::string& key, const std::string& value )
{
   if( !n )
      n = new tree_node( key, value, nullptr, nullptr );
   else
   {
      if( key == n -> key )
         return;   // Element present: No insert.

      if( key < n -> key )
      {
         insert( n -> left, key, value );
         return;
      }

      insert( n -> right, key, value );
      return;
   }
}
```