CSE687 Midterm #4

Name: ______Instructor's Solution___________ SUID: _____________

This is a closed book examination. Please place all your books on the floor beside you. You may keep one page of notes on your desktop in addition to this exam package. All Exams will be collected promptly at the end of the class period. Please be prepared to quickly hand in your examination at that time.

If you have any questions, please do not leave your seat. Raise your hand and I will come to your desk to discuss your question. I will answer all questions about the meaning of the wording of any question. I may choose not to answer other questions.

You will find it helpful to review all questions before beginning. All questions are given equal weight for grading, but not all questions have the same difficulty. Therefore, it is very much to your advantage to answer first those questions you believe to be the easiest.

1. Code analyzers often need to record type definitions, the files where they are found, and a list of all their namespaces. Provide a class declaration for a TypeTable that holds this information and implement as many of its methods you can with the time available to you.

```
template<typename TableRecord>
class TypeTable
{
public:
  using iterator = typename std::vector<TableRecord>::iterator;

  void addRecord(const TableRecord& record)
  {
    _records.push_back(record);
  }
  TableRecord& operator[](size_t n)
  {
    if (n < 0 || n >= _records.size())
      throw(std::exception("index out of range"));
    return _records[n];
  }
  TableRecord operator[](size_t n) const
  {
    if (n < 0 || n >= _records.size())
      throw(std::exception("index out of range"));
    return _records[n];
  }
  iterator begin() { return _records.begin(); }
  iterator end() { return _records.end(); }
private:
  std::vector<TableRecord> _records;
};
```

Since the table takes a template argument TableRecord we can store whatever information is appropriate for a specific application. That will need to hold a type, a name, a list or vector of namespaces, and other information the application needs, if any.

The information details discussed in the question, above, are illustrated in the complete solution in MTCode-S16, MT4Q1 project.

2. Why are the methods of a template class placed in the header file of its package?

   A C++ compiler can't generate code for a template class or function until the using application instantiates it with a known type. So templates are checked for syntax when compiled, deferring code generation until an application includes the template's header file and is compiled.

   The compiler does not have access to the template's implementation file (it only sees the included header files' text) so you must provide all the template implementation details in the header.

   This is the sequence of assertions that demonstrate the need:

   a. When templates are compiled their parameter types are not known and so no code can be generated. Only simple syntax checking takes place.
   b. Only when an application instantiates the template type or function does the compiler know the type. Then, if and only if, it has the complete definition of the template can it generate code for the instantiation.
   c. Since only header files are included by the application code, the entire definition of the template must be found in one of its included headers.

3. Suppose you are given a std::vector of pointers to structs where each struct identifies a person, having an entry for first name, last name, and city of residence. Write a lambda to display all the entries with duplicate last names, e.g., John Smith Syracuse, Bill Smith Albany, etc.

```
/*
 *  Note:  I'm finding people who live in the same city
 *         instead of having the same last name.
 *
 *         The code is essentially the same.
 *         Just substitute last name where I've used city.
 *
 *         I think this example is more interesting.
 */
struct Person
{
  std::string firstName;
  std::string lastName;
  std::string city;
};

using People = std::vector<Person*>;

void showPerson(Person person)
{
  std::cout << "\n  ";
  std::cout << std::setw(10) << person.firstName;
  std::cout << std::setw(15) << person.lastName;
  std::cout << std::setw(10) << person.city;
}

std::function<void(People&)> f = [](People& people)
{
  using City = std::string;

  std::unordered_map<City, People> collectDups;

  for (auto pPerson : people)
  {
    collectDups[pPerson->city].push_back(pPerson);
  }

  for (auto item : collectDups)
  {
    People temp = item.second;
    if (temp.size() > 1)       // move than one person in this city
    {
      for (auto pPerson : temp)
        showPerson(*pPerson);
    }
  }
};
```

4. State the Interface Segregation Principle and describe how to apply it in practice. Give an example discussed in class where ISP has been applied.

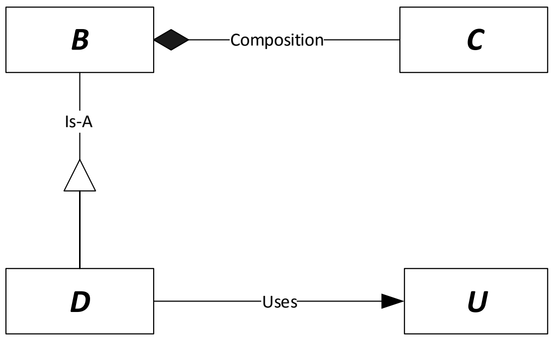
   “Clients should not be forced to depend on interface elements they do not use.”

   A very direct example of ISP is the Factory for AbstractProduct demo. There we provide a core interface IAbstractProduct and two interfaces that extend the language to invoke other behaviors, e.g., Interface1 and Interface2.

   The parser also uses ISP since the interfaces for rules and actions have been segregated. The parser only depends on IRule while the rules depend on IActon.

5. Given the compound object shown in the diagram, below, what happens when the following code executes?

```
B* pB = new D;
delete pB;
```



The composite object d constructed from D contains, within its memory footprint, an instance of the base class B which in turn contains an instance of the composed class C.

Construction of D will always cause B to be constructed (whether we explicitly invoke the B constructor or not) and construction of B will always cause C to be constructed (whether we explicitly invoke the C constructor or not).

Designers can always control how constructor proceeds by explicitly invoking constructions on bases and member instances in constructor initialization sequences.

For default construction there is no need to do that as the compiler will always invoke default constructors on bases and members.

For copy and move constructions the designer must explicitly invoke the base and member constructors.

See code in MT4Q1.cpp for these details.

///
// Complete solution
The first statement causes a default construction of D on the heap so we also get default construction of B and C. Assuming B has defined a virtual destructor, then the second statement calls D’s destructor. The destructor of D calls the destructor of B which, in turn, calls the destructor of C. If the B destructor is not virtual then only ~B() and ~C() are called.

Since U is used but not owned by D, none of the D operations directly cause U to be created or destroyed.
// End of solution ///

6. What do the functions std::thread::join() and std::thread::detach() do?

   a. std::thread::join() waits for the joined thread to exit its processing function when the underlying Windows thread becomes signaled, allowing the wait to return.

   b. std::thread::detach() disassociates the thread object from its underlying operating system thread. This allows the thread to continue without ever blocking the thread that called detach().

7. Given the class declaration below, describe the operation of each of the declared methods. A correct answer provides more information than just the operation names.

```
class X {
   public: X(); X(const X&); X(X&&); operator=(const X&); operator=(X&&); ~X();
   /* other methods that are not part of this question */
};
```

   a. X() is the default constructor for instances of X. It is used to initialize simple declarations with no arguments and arrays with no initializers.
   b. X(const X&) is the copy constructor for instances of X. Its function is to create a new instance with the same state as the source. It is used when objects are passed to funtions by value and\or returned by value when there is no move constructor. It is also used for explicit copy construction statements.
   c. X(X&&) is the move constructor for instances of X. Its function is to transfer ownership of resources from the source to a newly created instance. It is invoked when temporary X objects are returned from functions by value and any other time when the source of the operation is a temporary or in statements like X x2 = std::move(x1); The source is often invalid after the move operation.
   d. X& operator=(constX&) is a copy assignment operation. Its purpose is to make the state of an existing instance of X the same as the state of the source, and is invoked whenever an existing object is succeeded by the = operator.
   e. X& operator=(X&&) is a move assignment operator. Its purpose is to transfer ownership of the source's resources to an existing instance of X and deallocating the target's resources. It is invoked whenever the source is a temporary object or when we declare the operation like this: x2 = std::move(x1); As for move construction the source object is often invalid after the move.
   f. ~X() is the destructor of the X class. Its purpose is to deallocate resources used by an instance of the X class. It is invoked whenever an instance of X goes out of scope or an exception is thrown in the scope in which it is declared after the declaration. It is also invoked when an application calls delete on a pointer to an X instance on the native heap.