CSE687 Midterm #2

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. Describe the syntax and semantics for modern C++ casts.

   Answer:

   There are four modern casts:

   a. T t = static_cast<T>(s), where S s;
      This creates a new instance of t, casting from an instance of S. The cast operator is either generated by a built in C++ conversion rule for native types, or by a cast operator defined by the S class, e.g., S::operator T(); If no cast operator is defined by the class, but T has a promotion constructor taking an instance of S, that will be called. Otherwise, compilation fails.
   b. T* pT = const_cast(pConstT), where const T* pConstT;
      pT is typed as a pointer to non-const T, and may be passed to non-const functions. You should only do that if you are certain that the function will not change the state of the referenced instance of T.
   c. T* pT = reinterpret_cast<T*>(pS); where S* pS;
      The pointer pT references an instance s, referenced by pS, but treats it like an instance of T, e.g., uses T's type rules on the S instance. We normally only use reinterpret_cast to pack data into byte arrays for serialization or some similar process.
   d. D* pD = dynamic_cast<B*>(pB); where B* pB = new D;
      The pointer, pD, has the same address value as pB if D is derived from B. Otherwise its value is nullptr (0 in pre C++11 code). This supports accessing public members of D that are not inherited from B.

      Note: no cast is required to bind a pointer or reference of a derived type to a pointer or reference of its base type.

2. State the Dependency Inversion Principle (DIP). How do you support DIP in a C++ program?

   Answer:

   High level components should not depend on low level components. Instead, both should depend on abstractions.

   To break dependencies between a calling high-level component, H, and a low-level component, L, the low level component provides an **interface**, IL, and an **object factory** LFactory that has a creational function, IL* pL = LFactory::create(); Now, both H and L depend on the text of IL. The high level component also depends on the create() interface, but not on its implementation.

   If we build L as a **Dynamic Link Library** (DLL), then any change made to L to fix errors, performance issues, or add new functionality, will not affect H, as it only depends on the text of IL and the factory interface. When H is started, it will load L's DLL and does not have to be re-compiled or re-linked.

   Grading Note:

   Weak version of DIP uses interfaces and object factories to avoid recompilation when lower level changes.

   Strong version of DIP also implements lower-level components as dynamic link libraries to avoid relinking after change.

3. Describe the dynamic binding process used by C++ to implement polymorphism.

   Answer:

   Each class with at least one virtual function has a Virtual Function Pointer Table (VFPT).
   That table directs calls, to virtual functions that have not been overridden, to the base function code (usually[1]) at run-time, using dynamic dispatching using the VFPT.

   Any call to a virtual function that has been overridden will be directed to the derived class code, by VFPT dispatching process. It is the VFPT that allows a derived class to specialize the behavior of its base.

   That specialization is the essence of polymorphism.

   All calls to non-virtual functions are bound at compile-time and are not part of the class's polymorphic behavior, although they do participate in the Liskov Substitution process.

   Bjarne Stroustrup, the author of the C++ language, describes the use of template parameterization as static polymorphism. However, that does not involve any dynamic binding, e.g., all calls to template functions (either global or member) are bound at compile-time.

   Grading note:

   Dynamic_cast checks the inheritance chain to determine if a derived reference to an object is valid. It does not do any dynamic binding.

[1] In some special cases (only one class derived from an abstract base, for example) the compiler can bind the call at compile- time.

4. Write all the code for a class that holds package dependency information, including inserting and retrieving dependency relationships, and viewing dependency information.

   Answer:

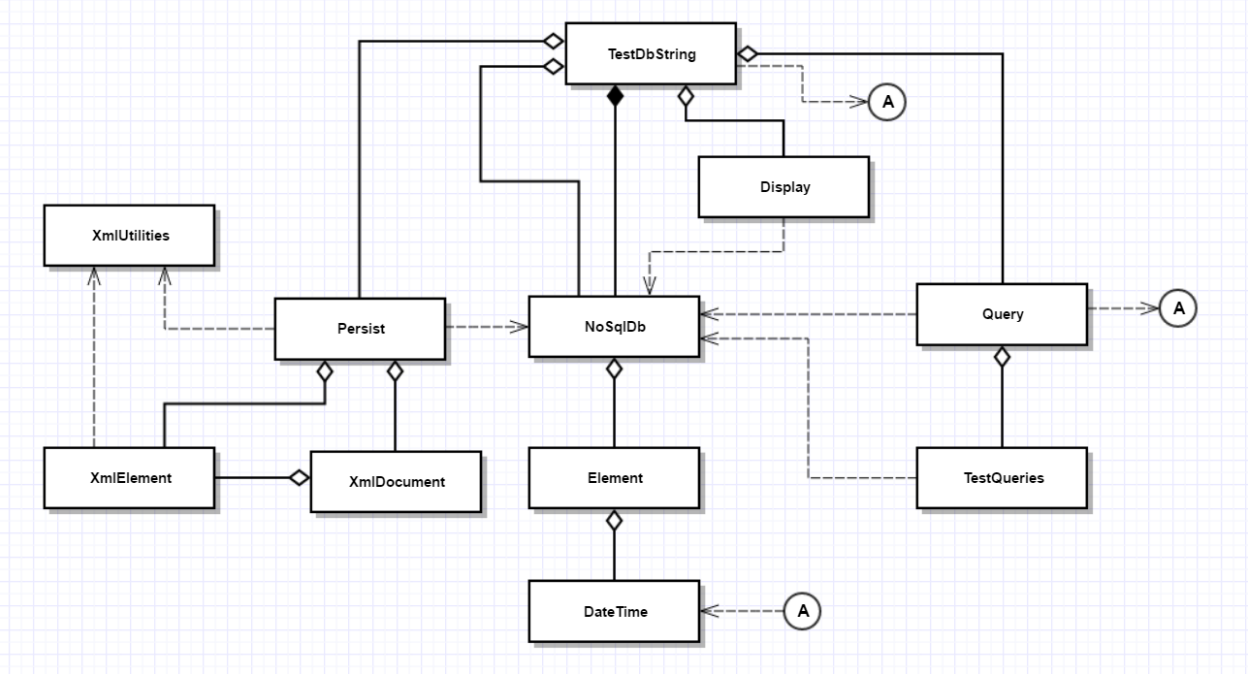
```
class PackageDependency
{
public:
  using File = std::string;
  using Dependencies = std::vector<File>;  // could use map, but lists will be short
  using DepMap = std::unordered_map<File, Dependencies>;

  PackageDependency& add(const File& parent, const File& child)
  {
    if (hasKey(parent))
      depRel[parent].push_back(child);
    else
    {
      Dependencies d;
      d.push_back(child);
      depRel[parent] = d;
    }
    return *this;  // allows chaining of adds
  }

  Dependencies get(const File& file)
  {
    if (hasKey(file))
      return depRel[file];
    else
    {
      Dependencies d;  // returning empty collection
      return d;
    }
  }
  void show()
  {
    DepMap::iterator iter = depRel.begin();
    for (iter = depRel.begin(); iter != depRel.end(); ++iter)
    {
      std::cout << "\n  " << iter->first;
      for (File f : iter->second)
        std::cout << "\n    " << f;
    }
  }
  void clear()
  {
    depRel.clear();
  }
private:
  bool hasKey(const File& f)
  {
    if (depRel.find(f) == depRel.end())
      return false;
    return true;
  }
  DepMap depRel;
};
```

It's ok to use NoSqlDb to implement this class, but don't re-implement the db. Just use as data member of this class to store and retrieve relationships.

You will find a solution using NoSqlDb in MT2Q4b.

5. Draw a class diagram for your implementation of Project #1.

   Answer:

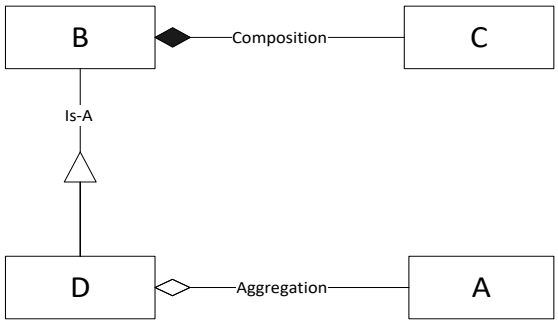


   Grading note:

   Exams, for this question, had problems with:

   a. Ownership – every class with no static members has to have an owner, with the exception of the Executive.
   b. Missing classes
   c. Incorrect or missing relationships
   d. Incorrect syntax
   e. Drawing package diagram instead of class diagram

6. Given the compound object described in the diagram, write a move assignment operator for the class D, assuming that classes B and C have correct move assignment semantics.



Answer:

This answer assumes that D has a std::string msg_ data member. Omitting that does not affect the accuracy of your answer.

```
//----< move assignment of D >-----------------------------------

D& D::operator=(D&& d)
{
  std::cout << "\n  move assignment of D";
  if (this == &d) return *this;
  B::operator=(std::move(d));
  msg_ = std::move(d.msg_);
  if (pA)
    delete pA;
  pA = d.pA;
  d.pA = nullptr;
  return *this;
}
```

7. Write all the code for a class that accepts a callable object that takes an instance of another callable object to run when its code completes. The second callable object is often referred to as a callback. Show how to use an instance of the class with a callable object and callback.

   Answer:

```
template <typename CO, typename CB>
class Invoker
{
public:
  void invoke(CO co, CB cb);
};

template<typename CO, typename CB>
void Invoker<CO, CB>::invoke(CO co, CB cb)
{
  co(cb);
}

#include <sstream>
#include <iostream>

std::string toString(size_t i)
{
  std::ostringstream out;
  out << i;
  return out.str();
}

int main()
{
  std::cout << "\n  MT2Q7a - Synchronous Invoker with Callback";
  std::cout << "\n ============================================";

  using CallBackType = std::function<void()>;
  using CallType = std::function<void(CallBackType)>;
  using Message = std::string;

  Invoker<CallType, CallBackType> invoker;

  CallBackType cb = []() { std::cout << "\n  this is a callback"; };

  for (size_t i = 0; i < 5; ++i)
  {
    Message msg = "Message #" + toString(i);
    CallType ct = [=](CallBackType cb) { std::cout << "\n  " << msg; cb(); };
    invoker.invoke(ct, cb);
  }
  std::cout << "\n\n";
}
```