CSE687 Midterm #1

Name: _____Instructor's Solution____________________

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 examinations 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 easiest.

1. What is polymorphism? Briefly describe all of the coding defects that have an adverse effect on polymorphism.

   Answer:

   Polymorphism describes an Object Oriented Language's ability to allow an interface pointer to bind to an instance of any class that implements the interface without any direct knowledge of the derived type. The function called through the interface pointer is determined by the class of the bound instance.

   This substitution capability is a very important mechanism for writing extendable code.

   Here is a list of the things that break polymorphism:
   - Hiding due to overloads across class scope or overloading virtual member functions
   - Overriding non-virtual member functions of a base class
   - Failing to provide a virtual destructor in a class that will be a base for inheritance
   - Use of default parameters in virtual functions that differ between base and derived
   - Failure to use initialization list when defining non-default constructors at all levels of the inheritance hierarchy

2. Write all the code for a Scope class that holds type, name, and complexity information so that each of these may be easily modified. Create a collection of Scope instances and show how you would sort that by complexity[1].

   Answer:

```
class Scope
{
public:
  std::string& name() { return _name; }
  std::string& type() { return _type; }
  size_t& complexity() { return _complexity; }

  static bool more(const Scope* pS1, const Scope* pS2);

private:
  std::string _name = "unknown";
  std::string _type = "unknown";
  size_t _complexity = 0;
};

inline bool Scope::more(const Scope* pS1, const Scope* pS2)
{
  Scope s1Test = *pS1;
  Scope s2Test = *pS2;
  return s1Test.complexity() > s2Test.complexity();
}

using Scopes = std::vector<Scope*>;
// define some scopes
Scopes scopes{ &s6, &s4, &s3, &s1, &s2, &s5 };
std::sort(std::begin(scopes), std::end(scopes), Scope::more);
```

[1] You may assume the existence of a std::sort function in the algorithm library that accepts two iterators describing the range to be sorted and a compare callable object that accepts two items from the sorted range.

3. Which parts of a class will the C++ compiler generate for you? How do you prevent automatic generation and when should you prevent it?

   Answer:

   The compiler will generate:

   - Default (void) constructor if no destructors are declared in the class
   - Copy constructor if implied by using code and not declared in the class
   - Assignment operator if implied by using code and not declared in the class
   - Destructor if not declared in the class
   - Address operator

   You prevent compiler generation of these functions by defining them for the class, by declaring them =delete, or by declaring them private and not implementing.

   If the class's bases and member data have correct copy, assignment, and destruction semantics then you should allow the compiler to generate them. Otherwise, you should prevent compiler generation.

   Note that it isn't useful to prevent generation of a destructor and not supply one as this will result in compilation errors when you declare an instance of the class in static or stack memory. It will only work if you create an instance on the heap and never call delete.

4. State the Open/Closed Principle and describe how you have used it in Project #2. If you have not used it, describe a useful way you could use it for that project.

   Answer:

   Software should be closed for modification and open for extension. This implies that the code has no latent errors nor performance problems. We normally soften this dictum by using interfaces and object factories that are invariant. Thus clients are not affected by any changes in the interface implementation to fix errors, enhance performance, or make subtle changes to its semantics.

   Another way of implementing OCP is to use template classes and functions. That allows the range of operation of the classes and functions to be extended by changing the type of the template parameter(s). Providing template policies and traits make these extensions particularly useful.

   In Project #2 you use the Parser. Parsing is extended by adding new rules derived from IRule and new actions derived from IAction. Also, the MTree and MNode classes are extended by supplying application specific template parameters without changing either class[2].

[2] You may have needed to change the MTree<T> class simply because it was incomplete. That does not alter the fact that using a template parameter in its design supports extension without modification.

5. Write all the code for an interface for the Scope collection you defined in your answer to the second question. Your interface needs to support sorting and comparing elements of the collection. Please provide an object factory as part of that interface. The comparison should return a value to indicate whether two scopes match.

   Answer:
   This was supposed to be a question about Interface for Scope class, not collection, as shown on this page. See next page for answer for Scope collection interface.

   The interface needs a virtual destructor, static creational function, and pure virtual declarations for each of the Scope class methods. I replaced the more function from Q2 with operator< to show how to use that for sorting.

   --------- Beginning of Answer – Interface for Scope class ---------------------------

```
struct IScope
{
  using Name = std::string;
  using Type = std::string;
  using Complexity = size_t;

  virtual ~IScope() {}

  static IScope* Create();
  static IScope* Create(const Name& name, const Type& type, Complexity comp);

  virtual Name& name() = 0;
  virtual Type& type() = 0;
  virtual Complexity& complexity() = 0;
  virtual bool match(const IScope* pS) = 0;
  virtual bool operator<(const IScope* pS) = 0;
};

// Scope class definition here

IScope* IScope::Create() { return new Scope; }

IScope* IScope::Create(const Name& name, const Type& type, Complexity comp)
{
  return new Scope(name, type, comp);
}
```

   -------- End of Answer – Interface for Scope Class ------------------------------

The Scope Collection is somewhat more complex but the interface is simple.

-------- Beginning of Answer – Interface for Scope Collection -------------------------

```
template <typename Compare> // bool compare(const Scope* pS1, const Scope* pS2)
struct IScopeCollection
{
  using Scopes = std::vector<IScope*>;
  using iterator = Scopes::iterator;

  static IScopeCollection<Compare>* create();
  virtual ~IScopeCollection() {}
  virtual iterator begin() = 0;
  virtual iterator end() = 0;
  virtual IScopeCollection<Compare>& add(IScope* pS) = 0;
  virtual void sort(Compare& compare) = 0;
};
```

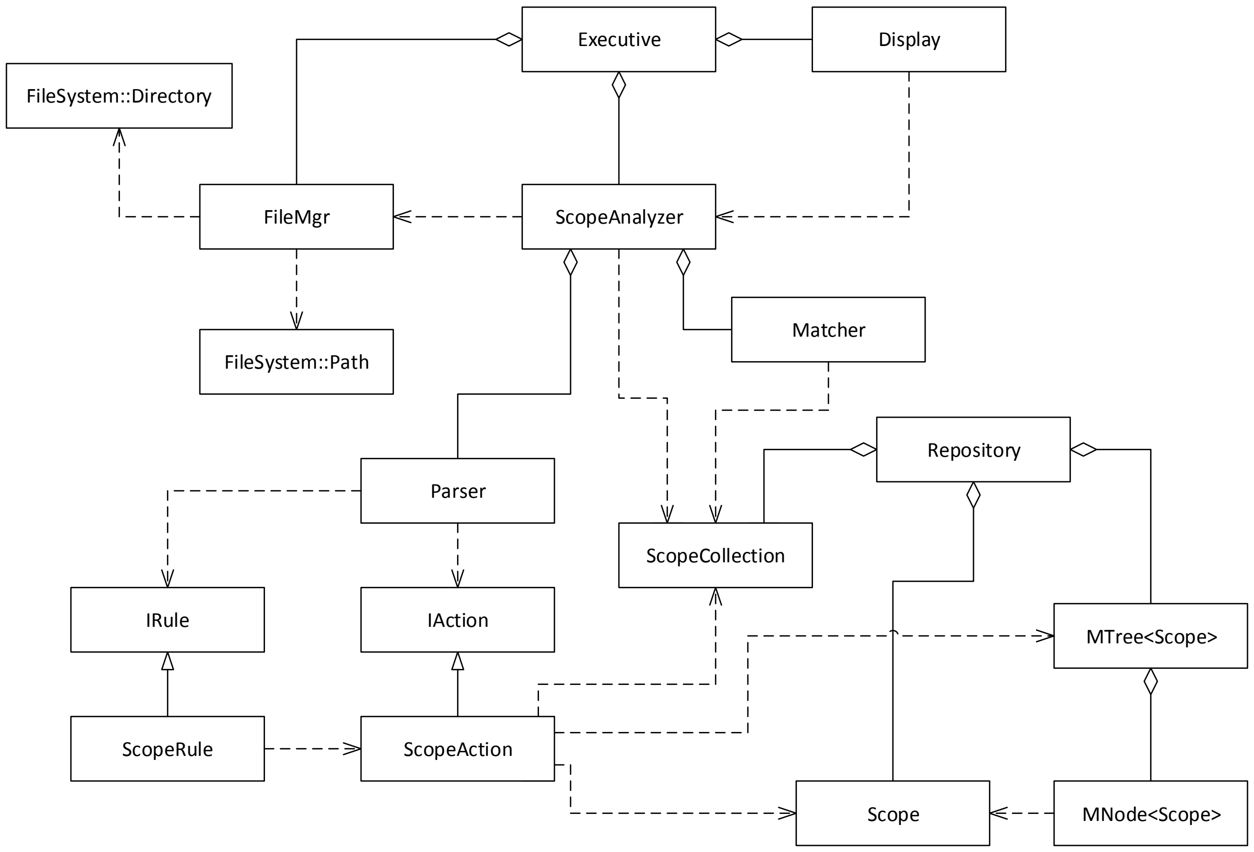
-------- End of Answer – Interface for Scope Collection ------------------------------

```
template<typename Compare>
class ScopeCollection : public IScopeCollection<Compare>
{
public:
  ScopeCollection() {}
  ScopeCollection(std::initializer_list<IScope*> lsp)
  {
    for (auto pScope : lsp)
      _scopes.push_back(pScope);
  }
  ScopeCollection<Compare>& add(IScope* pS)
  {
    _scopes.push_back(pS);
    return *this;
  }
  typename IScopeCollection<Compare>::iterator begin() { return _scopes.begin(); }
  typename IScopeCollection<Compare>::iterator end() { return _scopes.end(); }
  void sort(Compare& compare)
  {
    std::sort(
      _scopes.begin(),
      _scopes.end(),
      compare);
  }
private:
  std::vector<IScope*> _scopes;
};

template<typename Compare>
IScopeCollection<Compare>* IScopeCollection<Compare>::create()
{
  return new ScopeCollection<Compare>();
}
```

6. Draw a class diagram for your design of Project #2. You do not have to include classes associated with the parser.

   Answer:



ScopeAnalyzer gets a file specification from FileMgr and uses Parser to analyze its code scopes. The Parser's actions save that information in Scope objects placed in MNodes in the MTree and place a reference to each scope in a vector of scopes in the ScopeCollection object.

When all the files have been analyzed the Matcher requests ScopeCollection to sort its collection using a Matcher selected comparison object (not shown for lack of room). It then examines the scope collection for matches, looking at scopes that have sort proximity.

7. Describe the C++ memory model and how it affects your design of Project #2.

   Answer:

   C++ recognizes three forms of memory: static, stack, and heap.

   - **Static Memory:**
     Objects are created in static memory by preceding their declarations with the keyword "static". Static objects have the life-time of the program. A static object's constructor when static memory is initialized at the beginning of the program. Their destructors are called when the thread of execution leaves main.

   - **Stack Memory:**
     Objects are created in stack memory by declaring them as function parameters or declaring them locally in a function. Stack-based objects have a lifetime defined by the thread of execution residing within the scope in which the object is declared. Their constructors are called at the point of declaration and their destructors are called when the thread of execution leaves the scope in which they are declared.

   - **Heap Memory:**
     Objects are created on the heap by declaring them with the "new" keyword and destroying them with "delete". Heap-based objects live from execution of the statement using "new" to the statement using "delete". The new statement causes the allocated object's constructor to be called and delete invokes the object's destructor.

   Project #2 focuses on discovering and evaluating function scope structures which reside in stack memory. Scope information is stored in STL containers that use heap memory for data storage. We occasionally use references stored in static memory to provide global access to widely used objects. The Parser's repository can be accessed by this kind of static reference.

   Throughout the project design we make a number of decisions about how to store objects and data. We choose to use static, stack, or heap storage based on the required lifetime and accessibility of those objects and data.