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. What is the syntax for a header file and what code is allowed to be placed there?

   The header file (.h) is enclosed in preprocessor selection controls ensuring it is included only once in the composite code file passed to the compiler for translation, e.g.,

   #ifndef NAME_H, #define NAME_H, … #endif

   #pragma once is equivalent, but the statements above are preferred for portability.

   If it belongs to a package, a header file contains a prologue, manual page, and maintenance history.

   Header code consists of preprocessor include statements that declare types and functions whose names are used in this header. It also contains declarations of namespaces, classes, structs, and global functions. For templates and inline functions, the implementations must also appear in the header file if they are used by other packages.

   #includes of files that declare types and functions used only in the implementation file (.cpp) should be included there.

   Non-template and non-inline function definitions should not be placed in any header file. Also, no global data definition should ever appear in a header. If you must, you can declare global data with the extern keyword, indicating that this is just a declaration and a definition will be found in some implementation file (but you lose one point for each file with global data).

   Note that if the package uses classes and global functions not used by any other package, those should be placed in the implementation file with no declarations in the header. That holds even if they are templates.

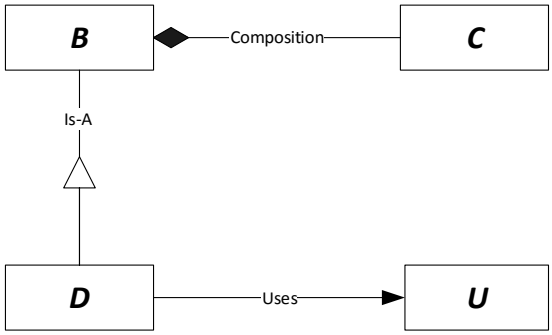
2. Write all the code for a parser rule that detects #include statements and an action that stores the file specifications from the includes in a data structure that associates it with the file currently being processed. You may assume that the name of that file is stored in a variable "currentFile".

```
////////////////////////////////////////////////////////////
// rule to detect #include

class Includes : public IRule
{
public:
  bool doTest(ITokCollection*& pTc)
  {
    size_t pos = pTc->find("#");
    if (pos < pTc->length())
    {
      if ((*pTc)[pos + 1] == "include")
        doActions(pTc);
    }
    return true;
  }
};
////////////////////////////////////////////////////////////
// save include file in Repo in
// std::un_orderedmap<File, std::vector<File>>

class SaveAssociation : public IAction
{
  Repository* p_Repos;
public:
  SaveAssociation(Repository* pRepos) : p_Repos(pRepos) {}
  void doAction(ITokCollection*& pTc)
  {
    size_t fileSpecIndex = pTc->find("include") + 1;
    if ((*pTc)[fileSpecIndex] == "<")
      ++fileSpecIndex;
    p_Repos->setFileAssociation(p_Repos->currentFile(), (*pTc)[fileSpecIndex]);
    std::cout << "\n  " << p_Repos->currentFile()
              << " includes " << (*pTc)[fileSpecIndex];
  }
};
```

Note:

setFileAssociation(…) and currentFile() are defined in Repository. See MTCode Supplement.

3. Given the compound object shown in the diagram, below, describe all the operations that occur when the object is returned by value from a function. What can the designer of the classes that are used to define the object do to make this operation efficient?



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 specific 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 unless those constructors are not declared by the class and compiler generated ones are correct.

See code in MT2Q3.cpp for these details.

///// minimum correct answer starts here ////////////////////

For return D by value D, B, and C are all moved, provided that the designer provided move constructors for all three. Otherwise any that don't provide move constructors are copied by their copy constructors. This assumes that correct constructor initialization sequences are provided for D and B. Since U is used but not owned by D, none of the D operations directly causes U to be created or destroyed.

//

Note that you should not return a reference to an internally created object as the reference will be invalid as soon as the return completes.

4. Write all the code for a function that runs a callable object asynchronously[1].

```
template <typename CallObj>
std::thread asynchRun(CallObj co)
{
  std::thread t(co);
  return t;
}

// the function above is a complete answer

void hello()
{
  std::cout << "\n\n  hello function:";
  for (size_t i = 0; i < 5; ++i)
    std::cout << "\n    running on thread with id = "
              << std::this_thread::get_id();
}

using namespace Utilities;
using Utils = StringHelper;

void main()
{
  Utils::Title("MT2Q4 - Running function asychronously");

  std::cout << "\n  main thread id = " << std::this_thread::get_id();

  std::function<void()> f =
  []()
  {
    std::cout << "\n\n  lambda:";
    for (size_t i = 0; i < 5; ++i)
      std::cout << "\n    running on thread with id = "
                << std::this_thread::get_id();
  };

  std::thread t = asynchRun(hello);
  t.join();

  t = asynchRun(f);
  t.join();

  std::cout << "\n\n";
}
```

I didn't explicitly ask for a function that accepts a callable object, so I have to accept any function that runs a callable object on a thread.

[1] By run asynchronously, we mean that the calling thread returns almost immediately without executing the called function's code directly.

5. State the Liskov Substitution Principle and describe the advantages afforded by its use. Give an example discussed in class where LSP has been applied.

   "Functions that use pointers or references statically typed to some base class must be able to use objects of classes derived from the base through those pointers or references without any knowledge specialized to the derived classes."

   Liskov Substitution supports:

   flexibility afforded by substitution

   ability to create reusable components that interact with applications without any application specific code.

   Liskov substitution allows code that uses base pointers to avoid the need to change when new derived classes are bound to the base pointers. An excellent example is the parser. The parser and all its associated infrastructure (Toker and SemiExp) need not change when we add new rules derived from IRule. And the rules do not need to change when we add new actions derived from IAction.

   Another example is the FileMgr, included with the MTCode Supplement. FileMgr defines Interfaces IFileEventHandler and IDirEventHandler. It does not need to change when new event handler objects are derived from those interfaces and registered for invocation. The FileMgr invokes whatever handlers have been registered without knowing or caring about the specific types of those objects and so uses no application specific code.

6. Which class operations will a standard conforming C++ compiler generate and when will that happen?

   The compiler will generate a default constructor when needed only if no constructors are declared for the class. It will be called to create instances and initialize arrays.

   The compiler will always generate a copy constructor, copy assignment operator, and destructor if those are not declared by the class and application code requests a copy or assignment (user code always requests destruction for local objects when they go out of scope).

   Move construction and move assignment will be generated only if both copy and move construction and assignment are not declared by the class. Also, neither of the move operations will be generated if the other is declared by the class.

   If any of these operations are declared delete then the compiler will not generate them, and if user code implies that operation, compilation will fail.

   Note that the compiler generated operations are executed on each base and member of the class. If All of the bases and members have correct copy, assignment, move and destruction semantics the operations are correct.

   Compiler generated operations are neither inherently shallow nor deep. They just call the corresponding operation on bases and members. If a member is a native pointer to something on the heap, then the resulting operation will be shallow and incorrect. If a member is a standard shared pointer, that has correct copy and assignment semantics, and the operations are deep and correct.

   For example, std::unique_ptr<T> has correct move construction and move assignment but does not have copy construction or copy assignment, so the copy operations will fail to compile in a class with a std::unique_ptr<T> member.

   std::shared_ptr<T> has correct copy construction and copy assignment semantics and those operations will compile in a class with a std::shared_ptr<T> member.

7. Discuss coding errors that affect the correctness of Liskov Substitution.

    a. Failing to provide a virtual destructor for a class that will be a base for inheritance. That causes incomplete destruction when the derived class is created on the heap and bound to a base pointer.
    b. Overloading of non-virtual functions across class inheritance scopes is an error in the derived class that causes hiding of the base overloads.
    c. Overloading of virtual functions in a base class is a base class error that causes hiding of the overloads in any derived class that overrides one of the overloads.
    d. Redefining a non-virtual base method in a derived class. That causes the base method to be called if a derived instance is bound to a base pointer.
    e. Using different default parameters in a virtual methods in a base and its derived class. That causes the default to be chosen by the type of the pointer, not the type of the object.