CSE687 Midterm #1

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.

Write all the code for a class that implements a node for an Abstract Syntax Tree for Project #2. Assume that all nodes are placed on the heap and should hold all the information needed to represent scopes for building the AST.

```
class ASTNode
{
public:
  using iterator = std::vector<ASTNode*>::iterator;
  using Name = std::string;
  using Type = std::string;
  using Line = size_t;
  iterator begin();
  iterator end();

  ASTNode();
  ASTNode::ASTNode(const Type& type, const Name& name, Line start, Line end=0);
  Name& name();
  Type& type();
  Line& startLine();
  Line& endLine();
  void addChild(ASTNode* node);
  std::vector<ASTNode*> children() { return _children; }
  void clear();
  size_t size();
  void makeChildAndPush(ScopeStack<ASTNode*>& ss);
  static void AddLineCountAndPop(ScopeStack<ASTNode*>& ss, size_t linecount);
private:
  std::vector<ASTNode*> _children;
  Name _name;
  Type _type;
  Line _startLine;
  Line _endLine;
};

ASTNode::ASTNode()
{
  _name = ""; _type = "anonymous"; _startLine = 0; _endLine = 0;
}

ASTNode::ASTNode(const Type& type, const Name& name, Line start, Line end)
{
  _name = name; _type = type; _startLine = start; _endLine = end;
}

std::string& ASTNode::name() { return _name; }
std::string& ASTNode::type() { return _type; }
size_t& ASTNode::startLine() { return _startLine; }
size_t& ASTNode::endLine() { return _endLine; }
ASTNode::iterator ASTNode::begin() { return _children.begin(); }
ASTNode::iterator ASTNode::end() { return _children.end(); }
void ASTNode::addChild(ASTNode* pNode) { _children.push_back(pNode); }

void ASTNode::clear() { _children.clear(); }
size_t ASTNode::size() { return _children.size(); }
```

```
void ASTNode::makeChildAndPush(ScopeStack<ASTNode*>& ss)
{
  ss.top()->_children.push_back(this);
  ss.push(this);
}

void ASTNode::AddLineCountAndPop(ScopeStack<ASTNode*>& ss, size_t linecount)
{
  ss.top()->endLine() = linecount;
  ss.pop();
}
```

2. Describe the syntax of a C++ package. Why do we build projects using packages?

   A C++ package consist of two files, a header and an implementation file.

   The header contains a prologue, manual page, maintenance history, and declarations of classes and global functions. For templates and inline functions, the implementations also appear in the header file. The header is enclosed in preprocessor selection controls ensuring it is included only once in the composite code file passed to the compiler for translation.

   The implementation file contains all the implementations of functions declared in the header if not template or inline. It also contains a test stub, enclosed with preprocessor selection controls. This allows the test stub, used for construction testing, to be excluded without changing any of the package code.

   We use packages to divide large programs into relatively small parts that are relatively easy to design, implement, and test. Other benefits of using packages are that dependencies are easier to control and communication becomes easier to manage.

3. Write the code for a function that will accept any callable object. Now define a callable object and show how the function can execute it.

```
template <class CallObj>
void Executor(CallObj& co)    // function accepting any callable object
{
  co();                        // here’s the invocation
}

void hello()                   // callable object
{
  std::cout << "\n  Hello from hello()";
}

class MyFunctor                // callable object
{
public:
  MyFunctor(const std::string& msg) : _msg(msg) {}

  void operator()()
  {
    std::cout << "\n  " << _msg.c_str() << " from MyFunctor";
  }
private:
  std::string _msg;
};

using namespace Utilities;
using Utils = StringHelper;

int main()
{
  Utils::Title("MT1Q3 - Executing Callable Objects");

  Executor(hello);             // function invoking callable object
  MyFunctor mf("Hi");
  Executor(mf);                // function invoking callable object

  std::string msg = "Hi again";
  Executor(                    // function invoking callable object
    [=]() { std::cout << "\n  " << msg.c_str() << " from lambda"; }
  );
}
```

4. State the Dependency Inversion Principle and describe the advantages afforded by its use. Give an example discussed in class where DIP has been applied.

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

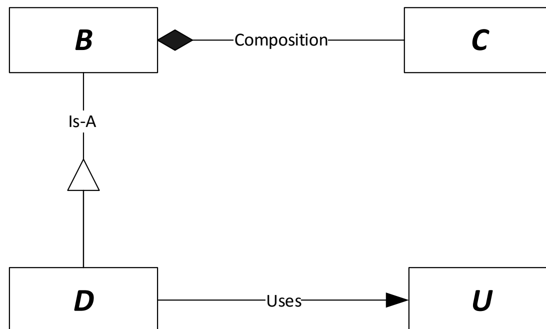
   DIP essentially implies that we use interfaces and object factories of implementation of components.

   The advantages of using DIP are that client code is isolated from any implementation changes made in components it uses if it accesses them using object factories and interfaces. It also means that parts that implement the interface can be interchanged without affecting other parts of the system.

   The Parser is a good example of DIP in action. Both the parser and its rules depend on the IRule interface. Also, both the rules and their actions depend on the IAction interface. All rules and actions are created by the ParserConfigToConsole factory.

   Another useful example is the FileMgr design suggested in class and illustrated in the FileMgr supplement to this Midterm's code.

5. Given the compound object described in the diagram, below, describe all the operations that occur when the object is created and when it is destroyed. Do you have any control over the specific operations invoked?



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).

When D's constructor starts, it immediately calls B's constructor and doesn't complete until B construction finishes. When B construction starts it immediately calls C's constructor and doesn't complete until C's construction finishes. The same sequence occurs on destruction.

Designers can always control how construction proceeds by explicitly invoking constructions on bases and member instances in constructor initialization sequences. They have no control over destruction other than the code that is supplied by each class since there is only one destructor defined by each class.

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 MT1Q5.cpp for these details.

Assuming a default construction of D we get default construction of B and C. When destroyed, the destructor of D calls the destructor of B which, in turn, calls the destructor of C.

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

6. Show all the ways you can think of to supply an argument to a thread to use while running.

```
void hello(const std::string& one, const std::string& two)
{
std::cout << "\n  arguments passed to hello are: " << one.c_str();
std::cout << " and " << two.c_str();
}

class MyFunctor
{
public:
MyFunctor(const std::string& one, const std::string& two)
: _one(one), _two(two) {}
void operator()()
{
std::cout << "\n  arguments passed to MyFunctor are: " << _one.c_str();
std::cout << " and " << _two.c_str();
}
private:
std::string _one;
std::string _two;
};

using namespace Utilities;
using Utils = StringHelper;

int main()
{
Utils::Title("MT1Q6 - Three ways to pass arguments to threads");

std::thread t1(hello, "one", "two");  // explicitly passed to thread constructor
t1.join();

std::thread t2(MyFunctor("one", "two"));  // passed as member data of functor
t2.join();

std::string one = "one";
std::string two = "two";
std::thread t3(                       // passed as lambda captured variables
  [=]() {
    std::cout << "\n  arguments passed to lambda are: " << one.c_str();
    std::cout << " and " << two.c_str();
  }
);
t3.join();
std::cout << "\n\n";
}
```

7. What are the differences in syntax and semantics between function overloading and overriding?

   Function overloading occurs when a function name is used more than once with different arguments in the same scope. These are treated as distinct functions by the compiler.

   Overloaded functions are resolved at compile-time using name mangling.

   Overloading allows the same conceptual operation to be invoked with different arguments. Class constructors are a good example.

   Function overriding occurs when a base class virtual function is redefined, in a class derived from the base, with the same argument types and the same or covariant return type. When invoked using a base class pointer, the overridden function belonging to the type of the referenced object is called.

   Overrides are resolved at run-time using the Virtual Function Pointer Table (VFPT).

   Overriding allows a derived class to modify the behavior of its base in specified ways, without modifying the base interface.