

## CSE687 Midterm #3

**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. State the Interface Segregation Principle (ISP). What benefits follow from designing your code to satisfy ISP? Have you seen any examples of its use in class?

Answer:

Clients should not be forced to depend on interfaces they do not use. Large interfaces lead to inadvertent coupling between clients that should be isolated.

If you segregate an object's interface into parts that are consistent with their expected use for each type of client, then if one client needs a change to their interface, that won't affect the design of other clients and will not require those clients to recompile.

We discussed the FactoryForAbstractProduct project where products could support any two of three possible interfaces and access to each interface was provided by cross-casting with the `dynamic_cast` operation. We also discussed the door control software where each door type was defined by mixing-in classes using multiple inheritance.

You could also use class template specialization to provide an interface appropriate for each type of client, e.g.:

```
template<typename Special, typename Core>
class applic { ... }
```

```
template<typename Core>
class applic<Client1Type, Core> { ... }
```

```
template<typename core>
class applic<Client2Type, Core> { ... }
```

Graph classes that provide a separate function or functor for each algorithm supports ISP by avoiding polluting the graph interface with algorithms that users may not need.

From one point of view Parser may also be consider an application of ISP as its interface is very simple, leaving the Rule and Action interfaces to other classes.

2. Write code to convert a string representation of an unspecified type to an instance of the type. For what types does your function achieve its purpose? When will it fail to do so?

Answer:

The conversion function below will succeed if and only if the type T provides an extraction operator. The resulting behavior will be consistent with the goal of the function if the extraction operator de-serializes an instance of the type. That is true for all the primitive types and for some user defined types as well.

```
template<typename T>
T FromString(const std::string& str)
{
    std::istringstream in(str);
    T t;
    in >> t;
    return t;
}

int main()
{
    Title("MT3Q2 - Converting from String");
    double d = FromString<double>("3.14159265");
    std::cout << "\n value of " << typeid(d).name() << " = " << d;

    int i = FromString<int>("10");
    std::cout << "\n value of " << typeid(i).name() << " = " << i;

    std::cout << "\n\n";
    return 0;
}
```

3. Given the following code:

```
class X : public Y {  
    public:  
        ...  
    private:  
        std::ostream& out;  
        std::vector<std::string> stuff;  
};
```

Write a copy constructor for the class X, assuming that out and stuff are its only data members. Are there any issues with the behavior of your constructor?

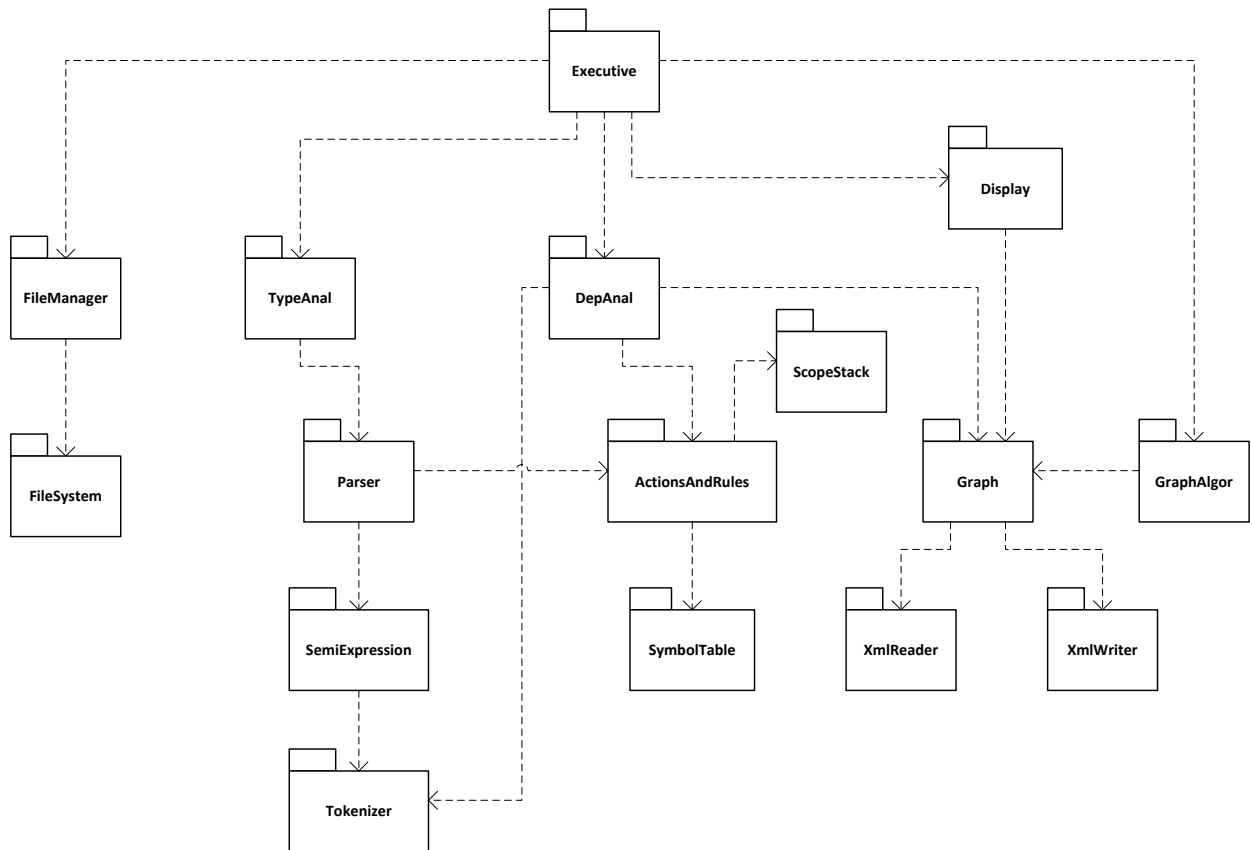
Answer:

The copy constructor, below, is equivalent to the compiler generated copy. Since the `std::ostream& out_` is a reference member the original and copy will share the same stream. This may or may not be desirable behavior. That depends on the application. For example, the original could close the stream while the copy was still trying to use it.

```
X::X(const X& x) : Y(x), stuff(x.stuff), out(x.out) {}
```

4. Draw a package diagram for your design of Project #2.

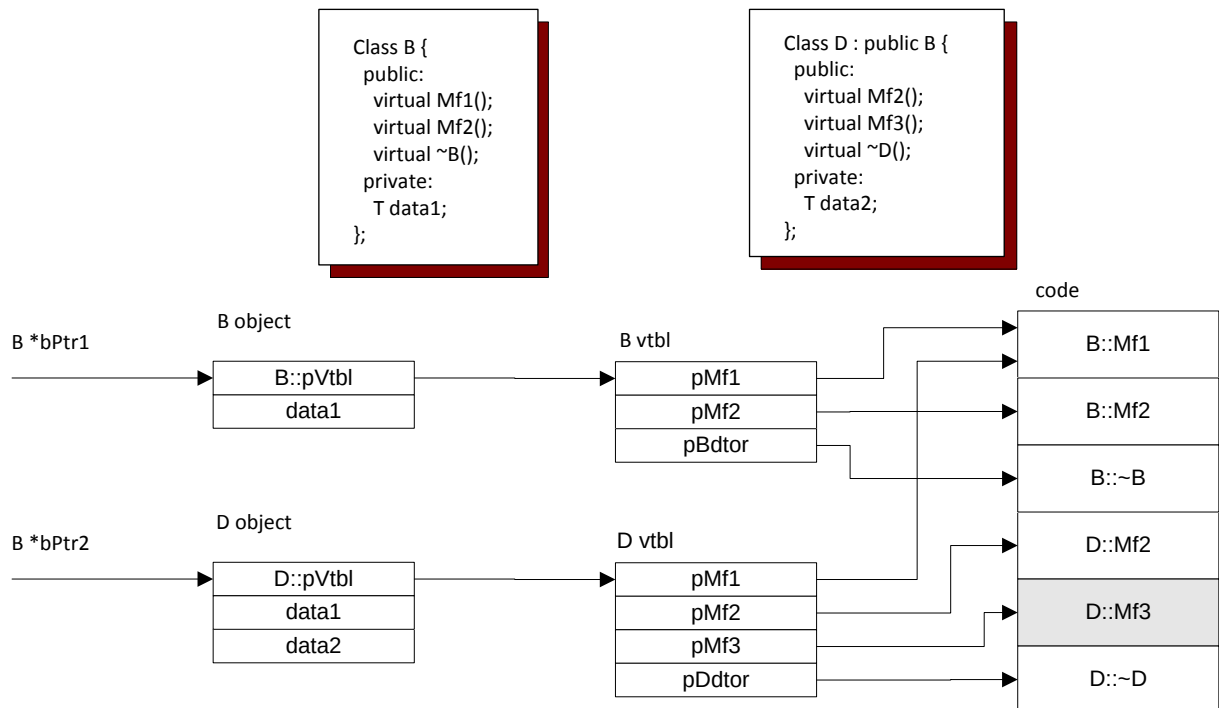
Answer:



5. What are virtual function pointer tables? When are they defined? What role do they play in Object Oriented Designs?

Answer:

Virtual function pointer tables are the core of the implementation of dynamic binding in C++. Each class with virtual functions has a table with pointers to each of its virtual function's code. If a derived class does not override its bases virtual function then the pointer binds to the base function. If the class does override the base virtual function then the pointer binds to the override. Dynamic binding, as described here, is the basis for Liskov Substitution and hence to polymorphic behavior, e.g., when a caller uses a pointer of type base to reference an instance of a derived class. The function called is determined by the type of the bound instance, not the caller's pointer.



6. Write all the code for a class that stores and displays file dependency information without using a Graph class.

Answer:

```
typedef std::string FileName;
typedef std::list<std::string> DepList;

typedef std::unordered_map<FileName, DepList> Dependencies;

class FileDependencies {
public:
    void Add(const FileName& src, const FileName& dst);
    void Show(std::ostream& out);
private:
    Dependencies deps;
};

void FileDependencies::Add(const FileName& src, const FileName& dst)
{
    Dependencies::iterator iter = deps.find(src);
    if(iter == deps.end())
    {
        DepList newDeps;
        newDeps.push_back(dst);
        deps[src] = newDeps;
    }
    else
    {
        iter->second.push_back(dst);
    }
}

void FileDependencies::Show(std::ostream& out)
{
    for(auto pair : deps)
    {
        std::cout << "\n " << pair.first << "\n ";
        for(auto file : pair.second)
            std::cout << file << " ";
        std::cout << std::endl;
    }
}
```

7. Explain how you would write an  $O(V+E)$ <sup>1</sup> function to find all vertices in an instance of Graph that have no parents. Remember that Graph's index operators return each vertex in the graph and the Vertex's index operators return each child reference. You don't need to write code to answer this question.

Answer:

Construct an array, with each entry initialized to zero, of size equal to the number of the graph's vertices, e.g., what `graph.size()` returns. Iterate down the graph's adjacency vector using the graph's index operator. At each vertex examine each edge, using the vertex's index operator. If an edge points to the *i*th location in the adjacency list then write 1 in the *i*th location of the array.

When you have visited the last child edge of the last vertex, the vertices with no parents are the ones that don't have an entry in the array, e.g., if the *j*th location of the array holds zero, then the *j*th vertex in the graph's adjacency list has no parents.

Notice that we have visited each vertex once and each edge once, so the complexity of this operation is  $O(V+E)$ .

---

<sup>1</sup>  $O(V+E)$  means that the time it takes the function to complete grows linearly with the number of vertices and edges in the graph.