

Section Handout

Discussion Problem 1: Muppet Inheritance

Consider the following set of class definitions (assume that all methods are **public**):

```
class Kermit {
    virtual void animal() = 0;
    void beaker() { muppet("Kermit::beaker"); animal(); }
    virtual void fozzie() { muppet("Kermit::fozzie"); rowlf(); }
    virtual void misspiggy() = 0;
    void rowlf() { muppet("Kermit::rowlf"); misspiggy(); }
    void muppet(string s) { cout << s << endl; }
};

class Statler : public Kermit {
    void beaker() { muppet("Statler::beaker"); rowlf(); }
    virtual void misspiggy() { muppet("Statler::misspiggy"); rowlf(); }
    void rowlf() { muppet("Statler::rowlf"); animal(); }
};

class Waldorf : public Statler {
    virtual void animal() { muppet("Waldorf::animal"); rowlf(); }
    void rowlf() { muppet("Waldorf::rowlf"); }
};

class Gonzo : public Kermit {
    virtual void animal() { muppet("Gonzo::animal"); rowlf(); }
    virtual void misspiggy() { muppet("Gonzo::misspiggy"); beaker(); }
    void rowlf() { muppet("Gonzo::rowlf"); }
};
```

Now consider the following function:

```
void muppetShow(Kermit *kermit) {
    kermit->fozzie();
}
```

What type of object can **kermit** legitimately address during execution? For each object type, list the output that would be produced by calling **muppetShow** against that type.

Discussion Problem 2: Scheme and Twisting Lists

Programming in our dialect of Scheme, write a function that takes a list of items and produces another list where each pair of elements has been exchanged. If there are an odd numbers of elements in the list, then just leave the last element in its original place. Here's a sample illustrating the functionality that must be managed by the definition of your twist function:

```
little-schemer 1> (twist (list))
```

```

()
little-schemer 2> (twist (list 1))
(1)
little-schemer 3> (twist (list 1 2))
(2 1)
little-schemer 4> (define mylist (list 1 2 3 4 5 6 7 8 9))
mylist
little-schemer 5> (twist mylist)
(2 1 4 3 6 5 8 7 9)

```

Discussion Problem 3: Scheme and Sorting

Write a Scheme function called **issorted**, which takes a list of integers and returns true if and only if the numbers are strictly increasing.

```

little-schemer 1> (issorted (list 1))
true
little-schemer 2> (issorted (list))
true
little-schemer 3> (issorted (list 1 2 3 4 5 6 7 8 9 10))
true
little-schemer 4> (issorted (list 1 3 2 4))
false
little-schemer 5> (issorted (list 8 3 1 4 9 1 11 32))
false

```

Lab Problem 1: JavaScript Object Notation Take II

For your one C++ lab problem this week, you're to rewrite the JSON parsing application you wrote three weeks ago, this time modeling the type hierarchy not using **unions**, but using inheritance. The problem domain should be familiar to you, but the implementation is new and leverages your newborn understanding of OOP and class design with parent and child classes.

There is a single file called **json-lite-inheritance.cpp** that currently processes integers just perfectly, but pretends that strings, Booleans, arrays, and dictionaries are all gibberish and ignores them. You should repurpose your previous implementation (or just rewrite it—it's really not that hard) to model the four outstanding types as subclasses of **JSONElement**, making use of the **virtual** keyword and runtime method resolution to implement an object-oriented JSON parser.

As part of the experiment, and after you've got everything working, you should remove the **virtual** keyword everywhere, provide dummy implementations instead of allowing **= 0** (which is off limits to non-**virtual** methods) and run the application to see what happens.