CS106X
Autumn 2015 Cynthia Lee

Section 8 (Week 9) Handout

Problem and solution authors include Jerry Cain and Aubrey Gress.

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.

Problem 2: Using Polymorphism with JavaScript Object Notation

JavaScript Object Notation, or JSON, is a popular, structured-data-exchange format that's easily read by humans and easily processed by computers. It's somewhat like XML in that both encode a hierarchy of information, but JSON is visually easier to take, and it's based on a small, fairly simple subset of JavaScript.

For the purposes of this week's discussion, we're going to assume that the only **primitive** types of interest are integers, strings, and Booleans. We'll pretend we're in a world without fractions, and characters can just be represented as strings of length one. We'll further simplify everything so that:

- All integers are nonnegative, so you'll never encounter a negative sign. We'll assume that all integers are small enough that they can fit into a C++ `int` without overflowing memory. (All JavaScript numbers are actually floating point numbers, but we'll ignore that for this problem.)
- Strings are delimited by double quotes (real JSON also allows strings to be delimited by single quotes as well, but we'll pretend that's not the case here).
- The only Boolean constants are `true` and `false` (all lowercase, and no delimiting quotes).

Each of the following represents a legitimate JSON literal:

```
1 4124 4892014 true false "CS106X" "http://www.facebook.com" "hello there"
```

More interesting are the two composite types—the array and the dictionary.

The array is an ordered sequence much like our `Vector` is, except that it's heterogeneous and allows the elements to be of varying types. The full arrays are bookended by `"["` and `"]"`, and array elements are separated by commas.

Here are some simple array literals, where all elements are of the same type:

```
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
[true, true, false, true]
["do", "re", "mi", "fa", "so", "la", "ti", "do"]
[["Rachel", "Finn"], ["Brittany", "Artie"], ["Puck", "Quinn"]]
```

But arrays can be truly heterogeneous, so that peer elements needn't be of the same type. That means the following also flies:

```
[1138, "star wars", false, [], [8, "C3PO", ["empire", [true], "R2D2"]]]
```

While the above array is contrived to illustrate heterogeneity, it's representative of the weakly typed programming languages that have blossomed over the past 20 years. (C++ is strongly typed, which is why everything needs to be declared ahead of time, and why any given C++ `Vector` can only store data of a single type.)

The JSON dictionary is little more than a serialization of a `Map`, save for the difference that the values needn't be all of the same type (heterogeneity once again). The dictionary literal is bookended by `"{"` and `"}"`, and individual key-value pairs are delimited by commas. Each key-value pair is expressed as a key (with double quotes to emphasize the requirement that it be a string [in our version, anyway]), followed by a colon, followed by the JSON representation of the key's value. Here are some JSON dictionaries to chew on:

```
{"Bolivia": "Sucre", "Brazil": "Brasilia",
"Colombia": "Bogota", "Argentina": "Buenos Aires"}
```

```
{"Hawaii": [808], "Arizona": [480, 520, 602, 623, 928], "Wyoming": [307],
"New York": [212, 315, 347, 516, 518, 585, 607, 631, 646, 716, 718, 845, 914, 917],
"Alaska": [907], "Louisiana": [225, 318, 337, 504, 985]}
```

```
{"Mike": {"Phone": "425-555-2912", "Hometown": "Midlothian, TX"},
"Jerry": {"Phone": "415-555-1143", "Hometown": "Cinnaminson, NJ"},
"Ben": {"Phone": "650-555-4388", "Hometown": "San Pedro, CA"}}
```

And just in case it isn't clear, arrays can contain dictionaries as elements. None of the above examples included any, because I hadn't introduced dictionaries by then.

Your job for discussion this week is to write a collection of functions that parse JSON, build in-memory versions of the JSON, print them out, and properly dispose of them. To get you started, I've provided implementations for the integers. Your job is to:

- read up on the `TokenScanner` class that comes with the CS106X library set as a C++ equivalent to Java's `StringTokenizer` class. Just view the documentation online by visiting the CS106X web site and then clicking on the **CS106X Library Documentation** link.
- understand why the in-memory versions of a JSON literal are actually trees when arrays and dictionaries come into play. It's that understanding that defends my decision to introduce JSON in a section handout on trees.
- complete implementations of the provided `parseJSON`, `prettyPrintJSON`, and `disposeJSON` functions. The implementation already handles the integer primitive type, and you're to extend them to handle Boolean, text, arrays, and dictionaries (for the latter two, paying careful attention to their heterogeneity). In the process you'll not only learn about JSON, memory, trees, pointers, and polymorphism, but you'll also gain some insight into how compilers go about their business when reading in larger program files.

"Pretty printing": in general it means that everything is printed inline with a reasonable amount of intervening whitespace between array and dictionary entries, and between key-value pairs. If you'd like, you can spend some time trying to emulate how full-blown pretty printing might work by trying to emulate the functionality produced at

```
http://www.cerny-online.com/cerny.js/demos/json-pretty-printing
```

IN SUMMARY:

This week, you're to rewrite a JSON parser, modeling the type hierarchy using inheritance.

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. Write code to model the four outstanding types as subclasses of `JSONElement`, making using 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.