Virtual Functions and C++ File I/O

Function overriding (review)

Consider the following classes, `Pet` and `Dog`:

```
class Pet {
     public:
          string name;
          void print(void);
};

void Pet::print(void) {
     cout << "Pet" << endl;
}

class Dog : public Pet {
     public:
          string breed;
          void print(void);
}

void Dog::print(void) {
     cout << "Dog" << endl;
}
```

We can store a `Dog` object in a `Pet` pointer, since all dogs are pets:

```
Pet *p = new Dog;
```

When we call `print`:

```
p->print();
```

The compiler looks at the variable type, which is a `Pet*`. Thus the function in `Pet` is called, which prints "Pet". This is very different from Java, which looks at what kind of object we have.

Virtual functions

To force the compiler to use the object instance when determining which function version to call, we can declare *virtual functions*.

Example

Here is our pet/dog example using a virtual function:

```
class Pet {
     public:
          string name;
          virtual void print(void);
};

void Pet::print(void) {
     cout << "Pet" << endl;
}

class Dog : public Pet {
     public:
          string breed;
          void print(void);
};

void Dog::print(void) {
     cout << "Dog" << endl;
}
```

Using virtual functions
Now, when we store a `Dog` in a `Pet*` and print the information:

```
Pet *p = new Dog;
p->print();
```

Since `print` is now virtual, the compiler sees what kind of object is stored in `p`. It contains a `Dog` object, so the compiler calls the version of `print` in the `Dog` class. Thus, this code prints "Dog".

Pure virtual functions
In Java, we have the idea of an *abstract method*. This means that we want to define what a method does, but we do not yet know how to implement it. In Java, the methods inside an interface are abstract methods because they have not been implemented yet.

The C++ equivalent to an abstract method is a *pure virtual function*. To do this, we declare a function to be virtual and then set its declaration =0. This states that we will not implement this function in the current class, but that we may implement it in a child class.

Any class that contains a pure virtual function is called an *abstract class*. We cannot instantiate objects with an abstract class type, but we can write child classes that extend abstract classes. These child classes must implement the pure virtual functions or they will also be abstract.

When a child class extends an abstract class, all the pure virtual functions from the abstract class are automatically virtual in the child class.

Example
Consider the following classes:

```
class Polygon {
     protected:
          int width, height;
     public:
          Polygon(int, int);
          virtual int area(void) = 0;
};

Polygon::Polygon(int w, int h) {
     width = w;
     height = h;
}

class Triangle : public Polygon {
     public:
          Triangle(int, int);
          int area(void);
};

Triangle::Triangle(int w, int h) : Polygon(w, h) {}

int Triangle::area(void) {
     return (width*height)/2;
}
```

Here, `Polygon` is an abstract class because it contains the pure virtual function `area`. Notice that `area` is not implemented in the `Polygon` class. We also defined a `Triangle` class that extends `Polygon`, and implements the `area` function.

Using an abstract class
We declare a pointer with an abstract class type:

```
Polygon *p;
```

We cannot create a `Polygon` object, since it is abstract, but we can store a `Triangle` object in our `Polygon` pointer (since all triangles are polygons):

```
p = new Triangle(2, 4);
```

When we print the area of the triangle:

```
cout << p->area() << endl;
```

The compiler calls the `area` function in the `Triangle` class, since `area` is virtual. Thus it prints 4, the area of the triangle.

Slicing problem

Recall the `Dog` and `Pet` classes from a previous example:

```
class Pet {
     public:
          string name;
          virtual void print(void);
};

void Pet::print(void) {
     cout << "Pet" << endl;
}

class Dog : public Pet {
     public:
          string breed;
          void print(void);
};

void Dog::print(void) {
     cout << "Dog" << endl;
}
```

Suppose we do the following:

```
Dog d;
Pet p;
d.name = "Fido";
d.breed = "Spaniel";
p = d;
```

It is all right to store `d` in `p`, since all dogs are pets. What happens when we do this statement is that the provided `operator=` function is called, which copies over all the `Pet` instance variables from `d`. If we then try to do:

```
//Compiler error!
cout << p.breed << endl;
```

We will get a compiler error because `p` is a `Pet` object that does not have a breed. Furthermore, if we do:

```
p.print();
```

Then "Pet" will be printed, even though the print function is virtual. When we do "`p = d`", the `Dog` object becomes a `Pet` object.

This is called the *slicing problem* – all of the "`Dog`" information is sliced off when we store it in a `Pet` variable. We can circumvent this problem by using pointers instead of regular objects – then, we won't lose any information. For example:

```
Pet *ppet;
Dog *pdog = new Dog;
pdog->name = "Fido";
pdog->breed = "Spaniel";

//Now ppet points to the Dog object
ppet = pdog;
```

When we do:

```
cout << ppet->breed << endl;
```

`ppet` references a `Dog` object, which has a `breed` instance variable. Thus this statement is fine, and prints "Spaniel". If we do:

```
ppet->print();
```

Again, `ppet` points to a `Dog` object. The `print` function is virtual, so the compiler calls the version in `Dog`. Thus, "Dog" gets printed.

C++ File I/O

C++ has its own file input and output classes, which work very similarly to the `cout` and `cin` objects. In C++ programs, it is better to use C++ file I/O than to keep using `fscanf` and `fprintf` from C.

The file I/O classes are:

- **`ofstream`** – extends the `ostream` class and handles output to files
- **`ifstream`** – extends the `istream` class and handles input from files

Recall that `cout` is an `ostream` object and `cin` is an `istream` object. Thus, these classes will work much like `cout` and `cin`.

To do file I/O in C++, first include the following information at the top of the file:

```
#include <iostream>
#include <fstream>
using namespace std;
```

Next, create either an `ofstream` or an `ifstream` object:

```
ofstream fileOut;
```

-OR-

```
ifstream fileIn;
```

Next, open a file with the "`open`" function. Thus function takes the name of the file you're opening. For example:

```
fileOut.open("output.txt");
fileIn.open("example.txt");
```

Now, we can check to see if the file was opened properly by calling the `is_open` function:

```
if (!inFile.is_open()) {
      //handle the error
}

if (!outFile.is_open()) {
      //handle the error
}
```

We are now ready to read or write from the file. To write something to a file, use <<. To read something, use >>. These work exactly like `cout` and `cin`:

```
//Prints "test" and a newline to output.txt
outFile << "test" << endl;

int num;
//reads a number from example.txt
inFile >> num;
```

If we're reading from a file, we may want to check to see if we've reached the end of the file:

```
while (!inFile.eof()) {
      //Keep reading from the file
}
```

Finally, once we're done using the file, we must close it:

```
inFile.close();
outFile.close();
```

Example (input)
For example, suppose we have an input file called data.txt that looks something like this:

```
5
1
2
3
4
5
```

The first line of the file is the number of remaining lines in the file. After that, there is one integer per line (and the total number of integers is whatever the first line indicated). We want to create a dynamic array to store these numbers. (We will not store the first value in the array, since it is just the size). Here's how:

```
ifstream inFile;
int *nums;
int size, i;

inFile.open("data.txt");
if (!inFile.is_open()) {
     throw "File not opened properly";
}

inFile >> size;
nums = new int[size];

i = 0;
while (!inFile.eof()) {
     inFile >> nums[i];
     i++;
}
inFile.close();
```

(Recall that when we read integers, and whitespace is skipped automatically.)

Example (output)
Next, suppose we want to create a file, output.txt, that looks something like this:

```
5
```

5
4
3
2
1

The first line is the size from data.txt, and the remaining lines are the reverse of the lines from data.txt. Here's how we would write this file (continuing from the example above):

```
ofstream outFile;
outFile.open("data.txt");
if (!outFile.is_open()) {
     throw "File not opened properly";
}

outFile << size << endl;
for (i = size-1; i >= 0; i--) {
     outFile << nums[i] << endl;
}

outFile.close();
```