Understanding operator<< and Inheritance

Overloaded operators are said to be "overloaded" because they define a new meaning for an operator for which the language/compiler has already assigned a meaning. Like overloaded functions, they can be overloaded many times; the only requirement is that at least one argument must be unique.

Arguments and Overloaded Functions

Suppose that a program defines two overloaded functions as follows:

```cpp
f(int x) {
    ...
}
f(double x) {
    ...
}
```

When the program calls one of the overloaded functions as `f(3.14)`; the compiler is able to determine that it is the second function that is being called based on the type of the parameter: 3.14 is a `double`. Similarly, if the program defines two overloaded functions:

```cpp
print(ostream& out, int x) {
    ...
}
print(ostream& out, double x) {
    ...
}
```

the compiler is able to determine that the call `print(cout, 5)` is referring to the first function based on the second argument: one unique argument is sufficient to distinguish between any number of functions with the same name).

Arguments and Overloaded Operators

Overloaded operators are really just overloaded functions, but they differ from "regular" functions by supporting a calling syntax that looks more natural. For example, the compiler always allows a programmer to add two integers using the "+" operator: `5 + 10`, which looks like the way that we would write the problem using "pencil and paper." By overloading operators for programmer-defined classes, it is also possible to write code that looks similar to the paper and pencil representation of problems based on that class. If `f1` and `f2` are instances of class `Fraction`, then their sum can be written with an overloaded `+` operator as: `f1 + f2`.

The inserter operator is always defined as a friend of some class. For example, if the inserter is defined in the `Fraction` class, its prototype is:

```cpp
friend ostream& operator<<(ostream& out, Fraction f)
```
(where the programmer is free to change the variable names \texttt{out} and \texttt{f} as he/she chooses). Friend functions are not members of the class in which they are declared as friends. Therefore, the left hand operand becomes the first functional argument and the right hand operand becomes the second argument. If \texttt{f} is an instance of the Fraction class, the inserter can be called as:

\begin{verbatim}
    cout << f;
    or
    operator<<(cout, f);
\end{verbatim}

The syntax used in the first call is the most common; after all, if we were going to use the syntax illustrated in the second example, we might as well name the function \texttt{print} or something similar. If a program includes multiple overloaded inserter functions, the functions are distinguished by their second argument.

\section*{operator\textless\textless and Inheritance}

(Ignore the keyword \texttt{in} preceding each argument in the diagram; it is an artifact of MS Visio, which was used to draw the diagram, and indicates that data is flowing into the function.) Each class in the UML class diagram at the right defines one private data member. Each overloaded inserter operator (\texttt{operator\textless\textless}) must print the data member in its own class but must also call the inserter of its immediate parent to print any inherited member variables. Calling one inserter function from within another inserter function demands that the parameter types in the function call exactly match the argument types in the corresponding function definition. To accomplish this, the second argument of the inserter function is cast into an instance of its parent so that it matches the argument list of its parent's inserter function. For example, the \texttt{Star} inserter operator is defined as:

\begin{verbatim}
ostream& operator<<(ostream& out, Star& s)
{
    out << (Actor &)*s << s.balance << endl;
    return out;
}
\end{verbatim}

The expression \texttt{(Actor &)*s} matches and becomes the second argument in the \texttt{Actor} inserter when the \texttt{Star} inserter calls the \texttt{Actor} inserter: \texttt{out} \texttt{\textless\textless} \texttt{(Actor &)*s} (which is equivalent to \texttt{operator\textless\textless(out, (Actor &)*s)}). The same "trick" allows the \texttt{Actor} inserter to call the \texttt{Person} inserter, chaining the inserter calls back to the top of the inheritance hierarchy.