Notes on Implementation of Serialization

ajd27

January 21, 2000

Our version of serialization should be able to use reflection but not depend upon it. The suggestion in Bushmann's Reflection pattern is to have the serializer query the variables in a class and save or retrieve them. This has been our basic notion, but Markus insists that we not break encapsulation. There are some variables which should be private. I am taking that reminder seriously and wonder whether we can take advantage of the machinery of introspection for private member variables.

The traditional model of serialization suggests you pass to a class a pointer to an istream or ostream.

```
void Subject::Write(ostream& out) {
    out << nAtoms;
    out << timestep;
}
```

We are suggesting that all variables which should be saved in order to save an objects state can also be public variables.

```
void Subject::Write(Writer& out) {
    out.WriteIntrospectiveObject(this);
}

void Writer::WriteIntrospectiveObject(Introspective *const Subj) {
    // For every variable in the class's ClassData, get it and
    // write it to disk.
    for_each(Subj.VarBegin(), Subj.VarEnd(), WriteVar);
}
```

The second method requires that all variables you want to save be publicly accessible. This may not be proper.

The DataFieldInfo object stores pointers to member functions which get and set variables. Those member functions can be private, and C++ will enforce the access rules as it should. We do not need to enforce them ourselves. On the other hand, there is no point in showing a member variable in a user-interface dialog box's edit text field if it would be an error to try to set it.

What use would it be to store pointers to private set member functions in ClassData if no one can set them? Maybe the introspective class, itself, could use its ClassData to save and read its state.

Chris made a comment about using the Memento design from Gang of Four which may help this issue. The Memento pattern suggests that a class can store its internal state in an opaque container which outside clients can save and store but not muck inside. Then the object can re-initialize from a Momento. It seems like it might be useful here, but we aren't as concerned with keeping the outside world from accessing our variables as we are concerned with how we can get access to our variables. That is, the DataFieldInfo objects which contain pointers to member functions will enforce access rules if those functions are private. How can one write code where Introspection helps a class internally call its own private member functions?

Here is how that code might look. It uses dynamic casting to get to the correct function pointers. The goal of this routine is to allow an object to call its own private member function so the call will succeed.

```
ListOfAtoms::read(istream &in) {
    // Get a list of our private data fields from somewhere.
    vector<DataField<int>* >& privateIntFields;
    privateIntFields = this->GetPrivateIntFields();
    int iValue;    // Read an int from the stream or file.
    in >> iValue;
    // Convert the DataField<int> to a DataFieldInfo<ListOfAtoms,int>
    DataFieldInfo<ListOfAtoms,int>* LOAFieldInfo;
    LOAFieldInfo = dynamic_cast<DataFieldInfo<ListOfAtoms,int>*>(
            privateIntFields[0]);
    // Get the pointer to our own member function and call it.
    void (ListOfAtoms::*pfSetInt)(int) = LOAFieldInfo.pfSetValue;
    (this->*pfSetInt)(iValue);
}
```

We had to cast the DataField to a DataFieldInfo because it is the DataFieldInfo which has member variable pointers to our private member functions. We would put them in the `DataField<int>`, but it doesn't know about ListOfAtoms. Stored in a DataField with the wrong type, the function calls could not succeed. Finally, this method ends by calling our own member function on our own variable, which was the goal.

Let's take a step back. We have been talking about using the same machinery of introspection for public and private variables. What if we didn't use all of it or if we changed it, maybe only for the private variables. The code above is difficult because it is difficult to get hold of the exact pointer to member function from the DataFieldInfo. What if the ClassData class knew about the ListOfAtoms (through templates)? Maybe a class can have a static `ClassData<ListOfAtoms>` member. This ClassData does not need to know about base classes because it is responsible only for private data. Having a little member class which

manipulates data reminds me of a Momento. It could return the pointers to get and set functions for ListOfAtoms itself.