MD is for Meta-Data

ajd27@cornell.edu

January 19, 2000

1 Introduction

This document describes a proposed structure for the molecular dynamics code in order to permit objects to describe their member variables and functions to client objects. We are trying to automate the interface to clients such as the graphical interface and serialization component. On a larger scale, this technique could be a precursor to defining how disparate simulation components talk with each other.

The big picture is that all information about classes and their methods are brokered by a singleton object called the Meta-Object Protocol. This introspection pattern comes from Buschmann et al. If a class wants to register its information, it requests the Meta-Object Protocol in the constructor and saves information through it. Client objects request an instance of the Meta-Object Protocol and ask what classes and variables are available.

The Meta-Object Protocol (MOP) manages a few types of objects. First is some object capable of instantiation other objects. It is called an Object Creation Manager and will be, for us, a Prototype Manager. (The Prototype Manager is not yet written.) The MOP keeps a map of information about each class. Each class has its own ExtTypeInfo structure which contains a pointer to the C++ Run-time Type Information (RTTI) type_info struct, the size of the class's object, a list of base classes of this class, and a pointer to ClassData. It is the ClassData that holds all information about member variables.

2 How to Preserve Type Information

2.1 What We Expose

Now that we have outlined the larger structure, let's see how and why to build it by focusing on implementation of ClassData. For the purposes of this discussion, our introspective object is a ListOfAtoms, and it may be derived from a ClassData or contain a pointer to a static member ClassData or have access to it from an external source (the MOP).

The ClassData object is responsible for storing variables by name and exposing their get and set functions. Let's explore how we might implement this
Figure 1: Overview of Introspection
by making a few errors along the way. The point is to problemmatize our facilities for introspection and other meta-data. That means the code you see below won’t be code actually used unless it is marked as such. These are examples here for discussion.

How would you expect to pass that information to ClassData from the constructor of ListOfAtoms? We could pass pointers to member variables, but we are concerned that the location of the member variables could change (really?) or that the public data passed to or stored from an object may not correspond exactly to a member variable. Therefore we want to store get and set functions for the object. You cannot store a pointer to a member function of an object in C++. You can store a pointer to a static function or a pointer to the member function of a class. That is, you can save &ListOfAtoms::GetNumber, which is effectively an offset from the ListOfAtoms this pointer so it could be used to refer to the member functions of any ListOfAtoms as shown below.

```cpp
int (ListOfAtoms::*pGetInt)() const = &ListOfAtoms::GetNumber;
ListOfAtoms* pAtoms = new ListOfAtoms();
int nAtoms = (pAtoms->*pGetInt)();
```

There is no cast in C++ which will allow you to recast a pointer to member function to a pointer to void or pointer to static function. That means we need to keep the correct type somehow.

The design pattern in Buschmann suggests saving function pointers as raw byte offsets from the this pointer. That is how I would handle the problem in C. It would avoid a lot of questions about how to handle type information. I would like to say that my search for an alternative is an altruistic effort to allow type information to improve program structure, but the truth is I just can’t figure out how to make C++ recast pointers, even with the aid of reinterpret_cast.

## 2.2 Objects Which are Functions

If we acquiesce to using typed pointers to member functions, one guess for how an object would store its variable information is just to add pointers to the member functions from its constructor.

```cpp
class ListOfAtoms {
  ClassData classData;
  ListOfAtoms() {
    classData.AddInt(&ListOfAtoms::GetNumber,
                    &ListOfAtoms::SetNumber, "nAtoms");
  }
};
```

This is a good start, but it would require that ClassData accept pointers to member functions of ListOfAtoms, and only ListOfAtoms. We do need to save the correct type somewhere. If we were to derive ListOfAtoms from a base type Introspective and write ClassData.AddInt to take pointers to member
Figure 2: The introspective subject can wrap its functions in a DataFieldInfo object without loss of type information, but in order to send that information to the ClassData, it must rely on the base class of DataField which knows only the kind of data stored and not where it came from.

functions of Introspective, then we would find (I did find) that they do not work as expected. The expression (pAtoms->*pfGetInt()), with pfGetInt a pointer to member function of Introspective, would convert pAtoms to an Introspective*. When you convert an object to its base class, you see an offset version of the this pointer, eight bytes in this case, so that the pointer to member function is now pointing eight bytes away from where it should point in the pAtoms struct. Ugly, no? So we keep the exact type of the member functions using templates.

We have two options for what we templatize. We could make ClassData a template with the argument ListOfAtoms. That might make sense if ClassData is a member variable of ListOfAtoms. This isn’t how the code currently works, but what would it look like?

class ClassData {
public:
    virtual int GetInt(string sVariableName) const;
    virtual void SetInt(string sVariableName, int iValue);
    virtual double GetDouble(string sVariableName) const;
    virtual void SetDouble(string sVariableName, double dValue);
}

// Intro stands for Introspective, the class which is exposing // its member variables.
template<class Intro> class TClassData : public ClassData {
public:
    void AddField(int (Intro::*pfGetFunc)() const,
                  void (Intro::*pfSetFunc)(int, string sName);  
...plus the virtual functions above.
The class could store a TClassData with the correct type information, and it would have to be derived from a ClassData in order for the world outside to talk with it. Or maybe it could include its class type as a typedef? Let’s return to our story.

The other way to use templates to encapsulate type information is to templatize the pointer to member functions in an object DataFieldInfo.

```cpp
template<class Intro> class DataFieldInfo {
  public:
    typedef int (Intro::*GETFUNC)(void) const;
    typedef void (Intro::*SETFUNC)(int);
    DataFieldInfo(GETFUNC pfGetInt, SETFUNC pfSetInt,
                   const string& sName);
    int Get() const;
    void Set(int iValue);
  private:
    GETFUNC pfGetValue;
    SETFUNC pfSetValue;
};
```

The idea now is that we could create one of these objects in ListOfAtoms() and pass it to the ClassData object.

```cpp
ListOfAtoms() {
    classData.AddInt(DataFieldInfo<ListOfAtoms>(
        &ListOfAtoms::GetNumber,
        &ListOfAtoms::SetNumber,"nAtoms"));
}
```

That might be fine except that there is a crucial object missing, this. Let’s say ClassData has a DataField object and wants to return its get function. Somewhere inside of DataFieldInfo, it has to express pAtoms->*pfGetValue. Again, there are two ways I see to do this. Either pass the object in the constructor of DataFieldInfo,

```cpp
ListOfAtoms() {
    classData.AddInt(DataFieldInfo<ListOfAtoms>(
        this, &ListOfAtoms::GetNumber,
        &ListOfAtoms::SetNumber,"nAtoms"));
}
```

or pass it in the get function

```cpp
template<class Intro> class DataFieldInfo {
  public:
    int Get(const Intro* const pObject) const;
    void Set(Intro* const pObject, int iValue);
    ...;
};
```
If the pointer to the ListOfAtoms object is inside the DataFieldInfo, then the DataFieldInfo is only good for referencing one particular object. The second method allows us to use the same class information for different instances of that class.

We have right now that ListOfAtoms passes to its ClassData a DataFieldInfo object which should be everything it ever needs to know about get and set member functions. How can ClassData handle that information? Can we generalize our DataFieldInfo from handling ints to other types?

We can guide the structure of ClassData by thinking of client requirements. If a client has a class ListOfAtoms, it needs to know the type of variables and what functions to call to return them. I do not believe there is a way to access general types and retain type information without using strong casts and offset pointers as discussed earlier. Java reflection allows objects to return variables of the basic builtin types or the general Object type. (Every class in Java is automatically derived from Object.) Here, we will expose ints and doubles and work from there. If we want to expose general types, we need to make a common base class, but that's for later.

There are two ways I can see to present type information from ClassData. One is to locate it in the DataFieldInfo as a typename (such as int or double).

```cpp
template<class Intro, class Element> class DataFieldInfo {
public:
  typedef Element value_type;
  ...
};
```

This is very much in line with generic functions and the STL. We could instead try a fat interface in the DataFieldInfo

```cpp
template<class Intro, class Element> class DataFieldInfo {
public:
  enum NumberType {INT, DOUBLE, SPATIALARRAY};
  typedef Element (Intro::*GETFUNC)() const;
  typedef void (Intro::*SETFUNC)(Element);
  DataFieldInfo(GETFUNC pfGetInt, SETFUNC pfSetInt,
                 const string& sName);
  NumberType GetType(void) const;
  Element GetInt(Intro* pObject) const;
  void SetInt(Intro* pObject, Element iValue);
  Element GetDouble(Intro* pObject) const;
  void SetDouble(Intro* pObject, Element iValue);
  Element GetSpatialArray(Intro* pObject) const;
  void SetSpatialArray(Intro* pObject, Element iValue);
private:
  GETFUNC pfGetValue;
  SETFUNC pfSetValue;
};
```
it would be good to think about these two methods.

The current version of the code allows ClassData to store lists of DataFieldInfo by giving DataFieldInfo a derived type. ClassData has a list of ints, doubles, etc.

    template<class Element> DataField {
        public:
            typedef Element (Intro::*GETFUNC)(void) const;
            typedef void (Intro::*SETFUNC)(Element value);
            Element Get(Introspective* pObject) const = 0;
            void Set(Introspective* pObject, Element value) = 0;
            string& GetName() const;
            void SetName(const string& sName);
        private:
            string name;
    }

    template<class Intro, class Element>
    class DataFieldInfo : public DataField<Element> {
        public:
            DataFieldInfo(GETFUNC pfGetInt, SETFUNC pfSetInt,
                           const string& sName);
            Element Get(Intro* pObject) const;
            void Set(Intro* pObject, Element iValue);
        private:
            GETFUNC pfGetValue;
            SETFUNC pfSetValue;
    };

With this setup, the client passes a pointer to its ListOfAtoms object to the Get function, and that pointer is recast to a ListOfAtoms inside DataFieldInfo<ListOfAtoms,int>.

3 What We Have Learned So Far

Austern explains that generic programming relies on templates because the depend on an objects behavior rather than its type. This is precisely how we want clients to see our objects, for what they get and set rather than who they are. That is why templates are an appropriate solution to retaining type information for this application.

We might have begun this project with a set of requirements:

• Information about a class's variables must be available to client objects outside the class.

• There must be a way to access all member variables of all base classes with one function call.
Figure 3: There are two ways to picture placing static class data in an object. A separate structure for each class makes communication among classes difficult while using a single static store in a base class replicates the functions of a Meta-Object Protocol in a less plastic configuration.

- We want one copy of class information for each class (not each object).
- Making member data static is attractive for a two reasons:
  - The class meta-data is visible inside the class, for instance in the debugger.
  - Static allocation immediately ensures there will be only one copy of class meta-data.

There are drawbacks to saving data as a static member. Two diagrams of how to store class meta-data statically are shown in Fig. 3. Using static class data makes initialization difficult and reduces function. If each class must construct its own data, then the code to do so needs to be in the class’s constructor. Then, once the data is in place, a derived object will have difficulty talking with its base class. If, on the other hand, a base class, `Introspective`, stores the class data, then it must store data on every introspective class in the whole simulation. It takes on the duties of the MOP, but inside a common base class instead of outside where we can use object composition, or, for instance, replace the MOP with a new version derived from the old one. Lastly, any attempt to save class data as a member variable prohibits native types like `int` and `double` from having class data. We don’t expect to treat `ints` and `doubles` as class objects, but the generalization might be useful.

It would be prettier to have `ClassData` store variable information on all variables on equal footing rather than saving `intVariables` and `doubleVariables`. Templates won’t solve that question because we are supposing the client does not know what kind of objects they are to see in a class, and compilers require that all template instances be known, or deductible, at compile time. We could put all the `DataField` objects in one array, but we couldn’t get them out, even treating each `DataField` as an STL container (with a single object).