Metaclasses and Reflection in C++

Preface

Over the last two years I've held several tutorials on meta-classes and reflection in C++. I use the term reflection here in its original sense, "looking back to oneself", nowadays sometimes called introspection. The more general process to allow modifications at class level at run-time is the old task to provide a meta-level, but is today sometimes (mis-)called "behavioural reflection" or "structural reflection".

In some sense this article presents work in progress, though it's going on now for nearly ten years. It's not the definitive meta-object protocol for C++, but more a presentation of lesser-known C++ techniques to solve some specific design problems.

If you have comments on these techniques, or proposals how the problems could be solved completely differently, or if you find errors in this article, I really appreciate your feedback to dv@vollmann.ch.

This document can be found on the web at http://www.vollmann.com/en/pubs/meta/index.html.

Some source code to illustrate the implementation of the ideas of this article can be found at http://www.vollmann.com/download/mop/index.html.

The code in this article is not completely identical with that source code (due to typographical reasons and compiler restrictions). So, it could well be that some errors crept into the code here; if you find them, please mail me as well.

Introduction

C++ is a strongly typed compiler language. Though not as strongly typed as ADA, a C++ compiler will complain if you try to assign an object of one type to an object of another type (if there is no acceptable conversion). Obviously, this requires that the compiler knows all available types. More specifically, all classes must be known at compile-time[1]. But sometimes, it would be quite handy to add new classes at runtime. And in some application domains, this is absolutely necessary.

A simple story

Let's look at a simple example: Susan, the manager of a local bookstore, wants to expand into the Internet. So she asks you to write a simple program

for an Internet bookshop. No problem for you. Part of your solution will probably look like the class model in Fig. 1.

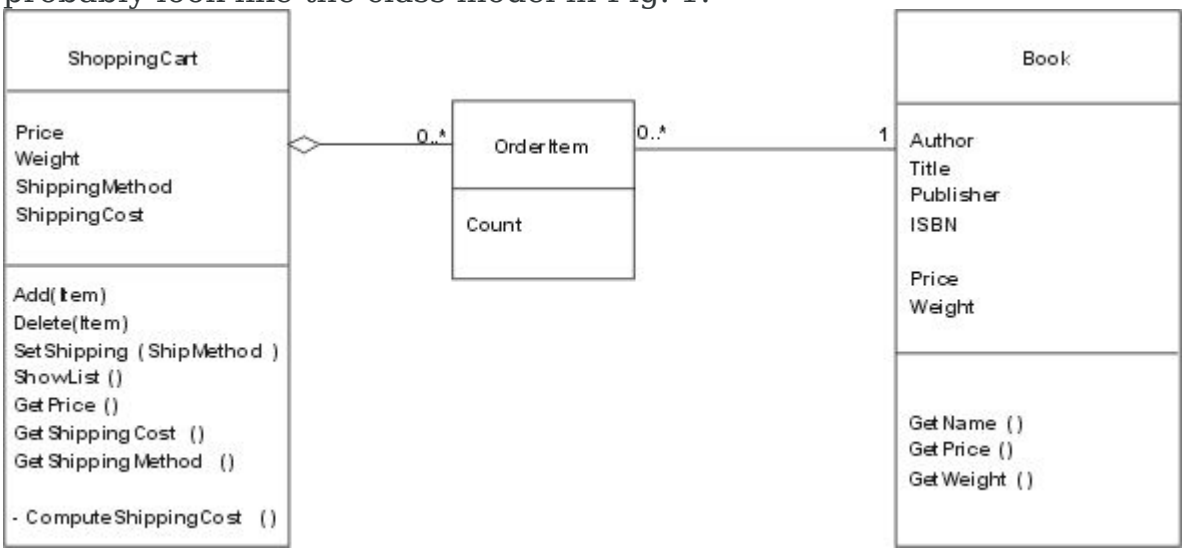

Fig. 1: Simple Shop Model

The implementation of this in C++ is straightforward. Here is the `Book` class:

```
class Book
{
public:
    Book(const string & author_,
         const string & title_,
         const string & publisher_,
         double price_,
         double weight_);
    string getName()
    {
        string name;
        name = author + ": " + title;
        return name.substr(0, 40);
    }
    double getPrice();
    double getWeight();
private:
    string author, title, publisher;
    double price, weight;
};
```

Your solution works, Susan is happy, and all is fine for a while...
But changes come on the Web in Internet time: the bookshop is a success and Susan decides to sell CDs as well. So you have to change your program. With object orientation, you can do this quite easily and your modified class model will look like Fig. 2.

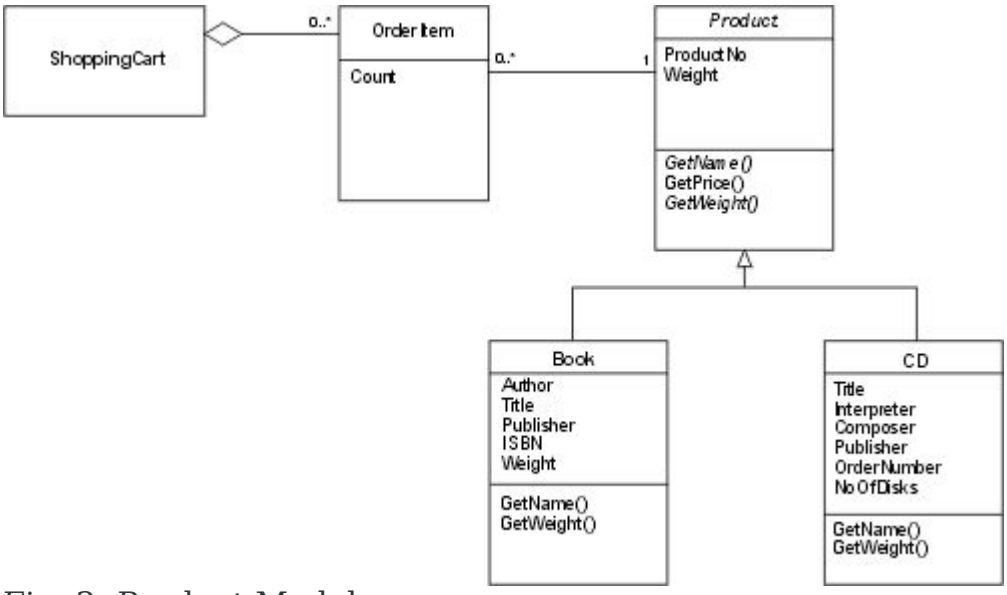

Fig. 2: Product Model

As you probably guessed, this was only the beginning. Some time later, Susan wants to sell Pop music accessories like T-shirts, posters, etc. as well.
Now it is clear that it is not acceptable to modify the source code of your program every time a new product category is introduced. So you start to think about the actual requirements, and find that you need to provide different interfaces for your `Product` class (Fig. 3): A simple interface for `ShoppingCart` providing `getName()`, `getPrice()`, and `getWeight()`. This is what you already have. Then you need a different interface for a general search machine[2], which must provide information like:

- what is the actual class of the object
- what attributes does that class have
- what are the actual values of these attributes for the object.

This is a classic reflection interface that gives you information about the properties of classes and objects.
But you also need a third interface for product maintenance that allows you to define new product classes, specify the attributes for them, create instances of these classes, and set the attribute values of these instances. Such an interface is called a "Meta-Object Protocol (MOP)" and thoroughly discussed in [1]. The reflection protocol is a subset of such a MOP.

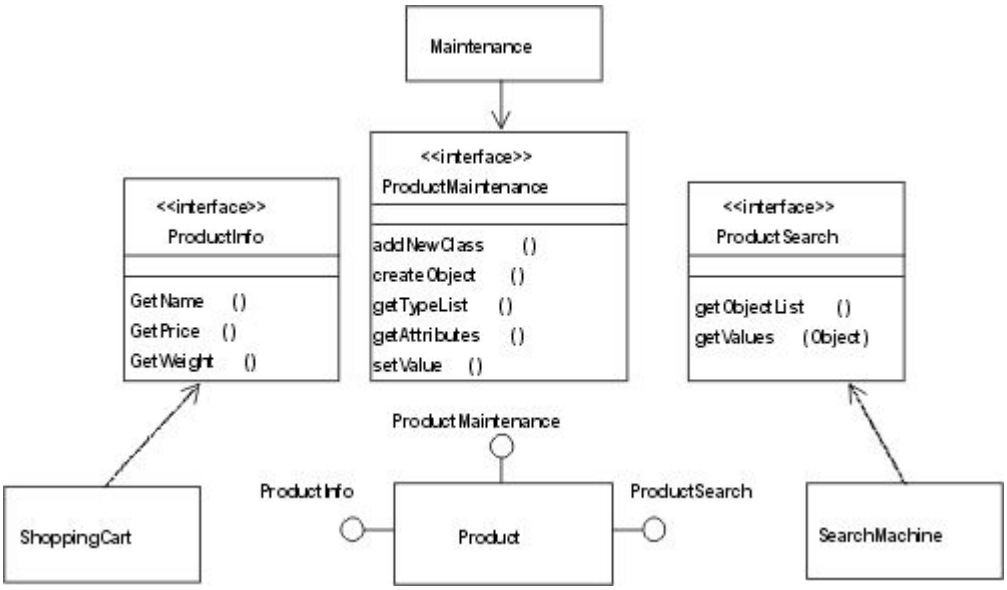

Fig. 3: A Better Model

Meta Classes for C++

What is the meaning of "Meta-Object Protocol"? Well, meta information is information about something else seen from a level beyond -- a meta level. So, information about the attribute values of an object, say `someBook.author`, is information on the object level. But information about the properties of the object itself, about its attributes, its structure, etc. is meta information.
In C++, this information is captured in the class definition for the object, so the class is a meta-object. And in C++, you have all the functionality of a MOP at class level -- which is at development time. But that level is not available at runtime: You cannot manipulate classes like objects, you cannot add new classes at runtime. The idea of a MOP is to collapse the meta-level (classes) and the object level (objects); i.e. make the class definitions normal objects and the object properties are normal attribute values of the class definitions that can be manipulated at runtime.
While languages like CLOS or Smalltalk provide this combined level directly, C++ as strongly typed compiler language has no such features. So, what can you do about it? The typical solution is to provide a MOP yourself, as proposed e.g. in [2] or [3].

MOP Overview

For simplicity, we ignore methods for now, so our MOP must provide:

- definition of new classes
- adding attributes to classes
- querying attributes of classes
- creating objects
- querying the class of an object
- setting attribute values of an object
- querying attribute values of an object
- deleting objects

If you use an old rule of OO design, you take all the nouns of the above requirements and make classes out of them. When you think about the "attributes" and "values" you have to decide whether they are typed. As the underlying language C++ is typed this should be mirrored in your design. Another question is about inheritance support. For our example with the Internet shop and a product hierarchy this would probably quite useful. So, a first class model is shown in Fig 4.

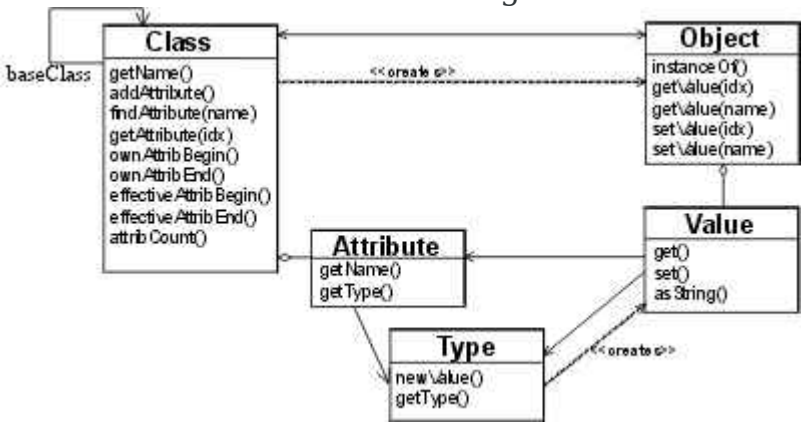

Fig. 4: MOP Class Model

Type

While it is useful to go top-down for a general overview, it's easier to start with the simple basic things for the details. So we'll first look at `Type`.
The main purpose of `Type` is to distinguish different kind of `Attribute`s. For this, a simple `enum` would suffice. But the idea of types is to have different kind of `Values` for different `Type`s, so the `Type` should create new `Value`s. So we put the `enum` inside the `Type` class, provide the `newValue()` method, and get the interface shown in Fig. 4.
Now for implementation. Though we don't look closer at `Value` for now, if we have different kind of values we probably need some base class for them. Let's call it "`BaseValue`", and `newValue()` can just return a pointer to `BaseValue`.
Now we know what to create, but how? While there are several patterns to implement polymorphic creation [4], the simplest one for our purpose is probably the *Prototype*, which can be easily implemented with a simple `static vector`[3].
Now we have everything to implement the entire class:

```
class Type
{
public:
    enum TypeT {stringT, intT, doubleT, unknownT};

    explicit Type(TypeT typeId_) : typeId(typeId_) {}

    BaseValue * newValue() const
    {
        return prototypes[typeId]->clone();
    }

    TypeT getType() const
    {
        return typeId;
    }
```

```
    static void init();
    {
        prototypes[stringT] = new Value<string>("");
        prototypes[intT] = new Value<int>(0);
        prototypes[doubleT] = new Value<double>(0);
    }

private:
    TypeT typeId;

    static vector<BaseValue *> prototypes;
};

vector<BaseValue *> Type::prototypes(Type::unknownT);
```

Attribute

As we have decided to create new `Value`s through `Type`, `Attribute` contains only a name and type, so here is its implementation:

```
class Attribute
{
public:
    Attribute(const string & name_, Type::TypeT typeId)
     : name(name_), type_(typeId)
    {}

    const string & getName() const
    {
        return name;
    }
    Type getType() const
    {
        return type_;
    }

private:
    string name;
    Type type_;
};
```

Classes

We can now finish the class (meta-) level of our model by looking at `Class` itself. As multiple inheritance is probably not an issue for our purposes, we have just one pointer to a base class (which can be 0). The more important question is the attribute list: Should it hold only the attributes defined for this class or should it include all inherited attributes? While for the actual object value access a complete list is more useful (and much faster), for class maintenance it might be important to know which attribute was defined in which class. So we just keep both. For the complete list, the order might be significant: should the own attributes come first or the inherited ones? In most illustrations the inherited attributes come first, so we keep this order as well.
What happens if the name of an attribute is the same as the name of an

inherited attribute? As C++ allows it, we can allow it as well (saving us some extra effort to check this), but in this case we must guarantee that on lookup of an attribute by name we get the most derived attribute. So, `findAttribute()` must do a reverse search. What shall we return from `findAttribute()`? The STL way would be to return an iterator, but for applications with GUIs (to create new objects and assign values to its attributes based on selection lists) an index-based access to the attributes will be more appropriate. So `findAttribute()` returns an index and `getAttribute()` takes an index and returns an `Attribute`. So the `Attribute` lists need to be an indexed containers, so we choose `vector`s for them.
A major purpose of `Class` is to create `Object`s from it, so it has a method `newObject()` which returns a pointer to an `Object`. Do we need to keep a repository with references to all created objects? For a full reflection interface we should do this. But for actual applications this is nearly never useful, as objects of the same class are created for completely different purposes. But do we need the repository for internal use? It depends on what we want to do with objects after they were created. This leads directly to another important decision: What do we do with already existing objects if we add a new attribute to a class? One option is to add this attribute to all existing objects and assign it a default value. The other option is to leave these existing objects and add the new attribute only to new objects. This leads to differently structured objects of the same class at the same time, and then we must add some version information to the objects. But there is a third option: To forbid the modification of a class definition once an instance of that class was created. This is the easiest option, so we adopt it for our MOP and add a flag `definitionFix`. With that flag, we can skip the object repository.
A last design question is when to add the attributes: At creation time of a class definition (through the constructor) or later (with a member function)? For different applications both options might be useful, so we'll provide two constructors and `addAttribute()`.
Now you can implement this[4]:

```
class ClassDef
{        //typedefs Container, Iterator for attributes
public:
    ClassDef(ClassDef const * base, const string & name_)
     : baseClass(base), name(name_),
       definitionFix(false)
    {
        baseInit();
        effectiveAttributes.insert(effectiveAttributes.end(),
                                   ownAttributes.begin(),
                                   ownAttributes.end());
    }

    template <typename iterator>
    ClassDef(ClassDef const * base, const string & name_,
             iterator attribBegin, iterator attribEnd)
     : baseClass(base), name(name_),
       ownAttributes(attribBegin, attribEnd),
       definitionFix(false)
    {
        baseInit();
        effectiveAttributes.insert(effectiveAttributes.end(),
```

```
                                    ownAttributes.begin(),
                                    ownAttributes.end());
        }

        string getName() const;
        Object * newObject() const
        {
            definitionFix = true;
            return new Object(this);
        }

        AttrIterator attribBegin() const;
        AttrIterator attribEnd() const;
        Attribute const & getAttribute(size_t idx) const;
        void addAttribute(const Attribute &);
        size_t getAttributeCount() const;

        size_t findAttribute(string const & name) const
        {
            // this does a reverse search to find the most derived
            AttributeContainer::const_reverse_iterator i;

            for (i = effectiveAttributes.rbegin();
                 i != effectiveAttributes.rend();
                 ++i)
            {
                if (i->getName() == name)
                {
                    return distance(i, effectiveAttributes.rend()) - 1;
                }
            }
            return getAttributeCount();
        }

    private:
        void baseInit()
        {
            if (baseClass)
            {
                baseClass->definitionFix = true;
                copy(baseClass->attribBegin(), baseClass->attribEnd(),
                     back_inserter<AttributeContainer>(effectiveAttributes));
            }
        }

        ClassDef const * const baseClass;
        string name;
        AttributeContainer ownAttributes, effectiveAttributes;
        mutable bool definitionFix;
    };
```

Values

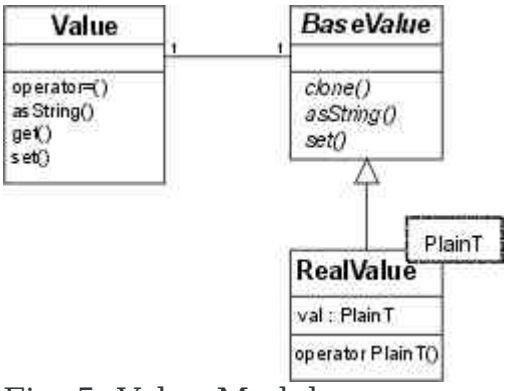

Fig. 5: Value Model

Before we can design `Object`, we have to think about `Value`. We need a common interface to manage them, which we already called `BaseValue`. But what interface do we need? The whole idea of `Value` is to store values, so we need a `set()` function. What parameter? The only thing we have is `BaseValue`, so that's the parameter type. Pass by value, by reference, or by pointer? Definitely not by value, as `BaseValue` is only an interface. On the other hand, what you pass is a value, so the parameter passing should be by value to let you pass temporaries. So one option would be to pass by const reference. But though this helps for the problem at hand, it doesn't cure the fundamental problem: you should have a value, but all you have is a polymorphic interface. The real solution here is the *pimpl* idiom, also known as *Cheshire Cat*, *Envelope/Letter*, or more generally *Handle/Body*. So we add a handle class, name it `Value`, and look at it later again. For now we're still at `BaseValue`.

We now have `set(Value)`, so what about `get()`? The return type of `get()` would be `Value`, and the implementation would look like:

```
Value BaseValue::get()
{
    return *this; // calls Value(BaseValue const &)
}
```

But that we can do directly, so `get()` doesn't make much sense.

What other `BaseValue` functions do we need? Values must be copied, so we add `clone()`.

That's all what we really need from a value, but we add `asString()` for convenience[5].

```
class BaseValue
{
public:
    virtual ~BaseValue(){}

    virtual BaseValue * clone() const = 0;

    virtual string asString() const = 0;
    // fromString()

    virtual void set(Value const & v) = 0;
    // no get()!
private:
    // Type info
};
```

RealValue

Now, as we have the interface, what about the implementation? We need values for `int`, `double`, `string`, ... And an `int` value must hold an `int`, a `double` value a `double`, etc. This looks like an opportunity for a template. So, let's define `RealValue<T>`, derive it from `BaseValue`, implement the inherited interface, and we're nearly done. But as `RealValue<T>` is just a wrapper around `T` with some additional functionality, but essentially still a `T`, we should provide conversion in both directions, by providing a converting constructor and a conversion operator.

```
template <typename PlainT>
class RealValue : public BaseValue
{
public:
    RealValue(PlainT v)
     : val(v) {}

    RealValue * clone() const
    {
        return new RealValue(*this);
    }

    string asString() const
    {
        ostringstream os;
        os << val;
        return os.str();
    }

    operator PlainT() const // conversion to plain type
    {
        return val;
    }

    RealValue<PlainT>::set(Value const & v)
    {
        val = v.get<PlainT>();
    }

private:
    PlainT val;
};
```

A note about `RealValue`: As we have conversion in both directions, we can use `RealValue<T>` like `T`:

```
RealValue<int> i = 1;
int j = i;
RealValue<double> d = i + 5.2 / (i*2);
cout << d << endl;
```

Nearly: the following doesn't work:

```
RealValue<string> name, author = "Bjarne", title = "The C++ PL";
name = author + ": " + title;
cout << name << endl;
```

The reason is that the compiler only applies one user-defined conversion, but for string literals, you need two: from `char const *` to `string`, and from `string` to `RealValue<string>`. If you want to work with `RealValue<string>` outside the MOP, you should define a specialization:

```
template <>
class RealValue<string> : public BaseValue, public string
{
public:
    RealValue(string const & s) : string(s) {}
    RealValue(char const * s) : string(s) {}
    RealValue() {}

    RealValue * clone() const
    {
        return new RealValue(*this);
    }

    string asString() const
    {
        return static_cast<string>(*this);
    }

    // no operator string(), conversion to base automatically

    void set(Value const & v)
    {
        string::operator=(v.get<string>());
    }
};
```

Note: Actually, its not really clean to derive `RealValue<string>` from `std::string`, but as long as you don't delete a `RealValue<string>` through a pointer to `string`, it will work.

Value handle

Now back to the handle class `Value`. As a handle class, it contains its body and cares for it. Its main job is to adopt/create and to delete its body. And it mirrors the interface of the body and forwards all messages. But it should also be a real value class, thus providing default and copy constructor and assignment. But how to implement the default constructor? As we don't know what type to create, we must create an empty handle without a body and check before forwarding if we actually have something to forward to. The assignment is essentially the `set()`, so we skip the `set()`.
Now let's come back to the `get()`. Of course, to return a `Value` or `BaseValue` doesn't make sense. But what about returning the `RealValue` or even the wrapped underlying value? That would be really useful, but for that we have to tell `get()` what we want as return type. So `get()` becomes a member template and so can return whatever is inside the `RealValue<>`.

```
class Value        // Value handle
{
public:
    Value(BaseValue const & bv)
     : v(bv.clone())
    {}
```

```
Value(Value const & rhs)
 : v(rhs.v ? rhs.v->clone() : 0)
{}

explicit Value(BaseValue * bv = 0)
 : v(bv)
{}

~Value()
{
    delete v;
}

Value & operator=(const Value & rhs)
{
    // this is not a typical pimpl assignment, but a set()
    if (v)
    {
        if (rhs.v)
        { // fine, all v's exist
            v->set(rhs);
        }
        else
        { // the other v doesn't exist, so we must delete our own :-(
            BaseValue * old = v;
            v = 0;
            delete old;
        }
    }
    else
    { // we don't have a v, so just copy the other
        v = (rhs.v ? rhs.v->clone() : 0);
    }

    return *this;
}

template <typename PlainT>
PlainT get() const
{
    if (v)
    {
        RealValue<PlainT> const & rv
            = dynamic_cast<RealValue<PlainT> const &>(*v);
        return rv;        // uses conversion operator
    }
    else
    {
        return PlainT();
    }
}

std::string asString() const
{
    if (v)
    {
        return v->asString();
    }
    else
    {
        return string();
    }
}
```

```
private:
    BaseValue * v;
};
```

Object

Finally we come to `Object`. Now, as we have everything else, an `Object` is mainly a container for its attribute values. To ease implementation, we will structurally mirror the attribute container in the class definition, so we use a `vector`. As we have so much effort invested in our `Value` handle, it would make sense to store that in the `vector`. But for future extensions it will be easier to have the `BaseValue` pointers directly available.
The constructor will create the values through the types of the attributes, so the only constructor takes a `ClassDef*`.
To set and get the values for the attributes, we provide two options: to specify the attribute by name and also by index.
For reflection purposes (as well as for internal implementation) we need a pointer to the class definition, but then we have it all:

```
class Object
{
public:
    explicit Object(ClassDef const * class_)
     : myClass(class_), values(class_->getAttributeCount())
    {
        buildValueList();
    }

    ClassDef const & instanceOf() const
    {
        return *myClass;
    }

    Value getValue(size_t attribIdx) const
    {
        return *values[attribIdx]; // calls Value(BaseValue &)
    }
    Value getValue(string const & attribName) const
    {
        size_t idx = instanceOf()->findAttribute(attribName);
        // should check for not found
        return getValue(idx);
    }

    void setValue(size_t idx, Value const & v)
    {
        values[idx]->set(v);
    }
    void setValue(string const & attribName, Value const &v)
    {
        size_t idx = instanceOf()->findAttribute(attribName);
        // should check for not found
        setValue(idx, v);
    }

private:
    typedef vector<BaseValue *> ValueContainer;
    void buildValueList()
```

```
        {
            ClassDef::AttrIterator a;
            ValueContainer::iterator i = values.begin();
            for (a = instanceOf()->attribBegin();
                 a != instanceOf()->attribEnd();
                 ++a, ++i)
            {
                *i = a->getType().newValue();
            }
        }

        ClassDef const * const myClass;
        ValueContainer values;
    };
```

Now the MOP is complete. Let's use it:
Creating the `Product` class:

```
ClassDef * product
     = new ClassDef(0, // no base class for Product
                    "Product"); // name of class
```

Adding attributes:

```
product->addAttribute(Attribute("Product Number", Type::intT));
product->addAttribute(Attribute("Name", Type::stringT));
product->addAttribute(Attribute("Price", Type::doubleT));
product->addAttribute(Attribute("Weight", Type::doubleT));
```

Creating the `Book` class with an attribute list:

```
list<Attribute> attrL;
attrL.push_back(Attribute("Author", Type::stringT));
attrL.push_back(Attribute("Title", Type::stringT));
attrL.push_back(Attribute("ISBN", Type::intT));

ClassDef * book
 = new ClassDef(product, // base class
                "Book",
                attrL.begin(), attrL.end());
```

Creating an object:

```
Object * bscpp(book->newObject());
```

Setting the values for the objects:

Set an int value by index (don't forget that index 0 is `ProductNo`):

```
bscpp->setValue(0, RealValue<int>(12345));
```

Same for a string value:

```
bscpp->setValue(4, RealValue<string>("Bjarne Stroustrup"));
```

Better way: set value by name this gives the most derived attribute:

```
bscpp->setValue("Title",
                RealValue<string>("The C++ Programming Language"));
bscpp->setValue("Weight", Value<double>(370));
```

Getting the values:

Display a book:

```
ClassDef::AttrIterator a;
size_t idx;
for (a = book->attribBegin(), idx = 0;
     a != book->attribEnd();
     ++a, ++idx)
{
    cout << a->getName() << ": "
         << bscpp->getValue(idx).asString() << endl;
}
```

and we get:

```
Product Number: 12345
Name:
Price:
Weight: 370
Author: Bjarne Stroustrup
Title: The C++ Programming Language
ISBN:
```

So, our MOP is complete. For our sample application, you have to add a class repository, some nice GUI to define classes and objects, creating the index for the search machine, provide an interface for `ShoppingCart`, but then you're done, and Susan is happy as she now can create her own new product categories at runtime.

Reflection for existing C++ classes

If you provide the interface for the `ShoppingCart` in our example, you'll find that it isn't so easy. If all classes are dynamic classes, all access must go through the MOP:
A `getName()` for `Book`:

```
string bookGetName(Object const * book)
{
    if (book->instanceOf().getName() != "Book")
    {
        /* throw some exception */
    }
    string name;
    //  name = book->author + ": " + book->title;  it was so easy...

    string author = book->getValue("Author").get<string>();
    string title = book->getValue("Title").get<string>();
    name = author + ": " + title;
```

```
    return name.substr(0, 40);
}
```

For a lot of applications, it would be useful to provide some classes of a hierarchy as C++ classes, e.g. `Product`, but still let the user add classes of the same hierarchy at runtime, e.g. `TShirt`. So, let's look at this. If we want access through our MOP to C++ classes, we need a `getValue()` to which we can give the attribute we want to access at runtime. So, here it is:

```
Value getValue(Object *o, MemberPointer mp)
{
    return o->*mp;
}
```

The magic lies in `'->*'`: This is the *pointer-to-member* selector of C++.

Pointer to member

You can imaging a pointer-to-member in C++ as an offset[6] from the base address of an object to a specific member. If you apply that offset to such a base address, you get a reference to the member (Fig. 6). But as a pointer-to-member is a normal data type in C++, you can store them in containers, pass them to functions, etc. Thus, you can write the function above, building the fundamental base for our combination of C++-classes and runtime-classes.

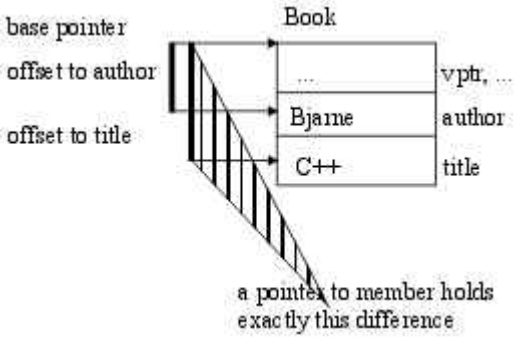

Fig. 6: Pointer to Member

Let's look at some details of pointer-to-members. As an example, we use the following class:

```
class Product
{
// ...
protected:
    RealValue<double> price;
};

class Book : public Product
{
public:
    // ...
private:
    RealValue<string> author, title;
    RealValue<double> weight;
};

Book b, *bp;
```

A pointer-to-member is a type that is derived from two other types: The type of the base object (`Book` in our example) and the type of the member (`RealValue<>`). The type decorator for a pointer-to-member is '`::*`', so let's define two variables with initialization:
`RealValue<string> Book::* bookStringMemPtr = &Book::author;`
`RealValue<double> Book::* bookDoubleMemPtr = &Book::weight;` The pointer-to-member selector comes in two variations: as '`.*`' you can apply it to references of the class and as '`->*`' it takes a pointer. It is a binary operand, as left operand it takes a reference (or pointer) to an object and as right operand a pointer-to-member. So, with the above definitions, you can do things like

```
b.*bookStringMemPtr = "Bjarne Stroustrup"; // assigns b.author

bookStringMemPtr = &Book::title;

bp->*bookStringMemPtr = "The C++ Programming Language"; // assigns b.author
```

Of course, as `title` is a private member of `Book`, the assignment of the pointer-to-member must be at the scope of that class. But the pointer-to-members themselves can be used even if you don't have access privileges to the members (as long as you have access to the pointer-to-member).

pointer-to-member Types

A word about the types: `RealValue<double> Book::*`, `RealValue<double> Product::*`, and `BaseValue Product::*` are different types. But are there conversions? The C++ standard provides a conversion from a pointer-to-member of a base class to a pointer-to-member of a derived class. That makes sense: You can apply an offset to a member of a base class to the base address of a derived object as well, as the base is a part of the derived object[7]. So you can assign `bookDoubleMemPtr = &Product::price;` as the `price` is part of each `Book` instance.
The other way around it obviously doesn't work, you couldn't initialize `RealValue<double> Product::* &Book::author;` as `author` is not a member of each instance of type `Product`.
But the standard does not provide a conversion from `RealValue<double> Book::*` to `BaseValue Book::*`, though it would be save: If the result type of the pointer-to-member selector is a reference to a derived class, it can be safely converted to a reference of a respective base class, so it would also be safe to let the compiler do the conversion automatically and therefore also convert the pointer-to-members themselves. As already mentioned, the standard doesn't provide (implicit) and even doesn't allow (explicit through `static_cast`) that conversion, probably because the committee didn't see any use for pointer-to-data-members at all (see [5]), and for pointer-to-member-functions that conversion really doesn't make sense.
The problem for us is: we need that conversion. We want to keep pointer-to-members to all members of a class in one common container, but what could be the type of that container's elements? One option would be to force the conversion through a `reinterpret_cast`, but the only thing you can safely do with a `reinterpret_cast`ed thing is to `reinterpret_cast` it back, and for that you have to store the original type as well. So we use another option: we just

define the conversion! But as C++ doesn't allow you to define your own conversions to compiler-provided types (and in this sense the pointer-to-members are compiler defined, though the involved single types like `Book` or `BaseValue` are user-defined), we have to define wrapper classes around them.
Here's the implementation:

```
template <typename BaseType, typename BaseTargetType>
class MemPtrBase
{
public:
    virtual BaseTargetType & value(BaseType & obj) const = 0;
    virtual BaseTargetType const & value(BaseType const & obj) const = 0;

protected:
    MemPtrBase() {}
    virtual ~MemPtrBase() {};

private:
    MemPtrBase(MemPtrBase const &);
    MemPtrBase & operator=(MemPtrBase const &);
};

template <typename BaseType, typename BaseTargetType, typename TargetType>
class TypedMemPtr : public MemPtrBase<BaseType, BaseTargetType>
{
public:
    TypedMemPtr(TargetType BaseType::* ptr)
     : p(ptr)
    {}

    BaseTargetType & value(BaseType & obj) const
    {
        return obj.*p;
    }

    BaseTargetType const & value(BaseType const & obj) const
    {
        return obj.*p;
    }

private:
    TargetType BaseType::* p;
};

template <typename BaseType, typename BaseTargetType>
class MemPtr // this is a handle only
{
public:
    template <typename BaseType2, typename TargetType>
    explicit MemPtr(TargetType BaseType2::* ptr)
     : p(new TypedMemPtr<BaseType, BaseTargetType,
        TargetType>(static_cast<TargetType BaseType::*>(ptr)))
    {}

    ~MemPtr()
    {
        delete p;
    }

    BaseTargetType & value(BaseType & obj) const
    {
        return p->value(obj);
```

```
        }

        BaseTargetType const & value(BaseType const & obj) const
        {
            return p->value(obj);
        }

    private:
        MemPtrBase<BaseType, BaseTargetType> * p;
    };
```

Some notes to the code: `BaseType` is used for the class to which a pointer to member is applied (e.g. `Book`), `TargetType` is the result type to which a pointer-to-member points (`RealValue<double>`), and `BaseTargetType` is the base class of `TargetType` (`BaseValue`). `MemPtrBase<>` is the common base class as we need it (e.g. `MemPtrBase<Book, BaseValue>`, which stands for `BaseValue Book::*`), `TypedMemPtr<>` hold an actual C++ pointer-to-member (`TypedMemPtr<Book, RealValue<double> >`), and `MemPtr<>` is a handle class around `MemPtrBase<>` to store them in a container.
Here, the actual access function is the `value()` member function. If you want, you can add a global operator '`->*`' (as template function), but you can't provide the operator by a member function (as the left operand is not the class instance), and you can't overload '`.*`' (this is one of the few non-overloadable operators).
The `MemPtr` constructor is a member template with two template parameters: a `BaseType2` and the `TargetType`. The second one is clear as it defines the actual `TypedMemPtr` to be constructed, but the `BaseType2` is not so obvious. If we omit the `BaseType2`, so only having

```
    template <typename BaseType, typename BaseTargetType>
    class MemPtr // this is a handle only
    {
    public:
        template <typename TargetType>
        explicit MemPtr(TargetType BaseType::* ptr)
         : p(new TypedMemPtr<BaseType, BaseTargetType, TargetType>(ptr))
        {}
        // ...
    }
```

and then we try to create a

```
    MemPtr<Book, BaseValue> mp2(&Product::price);
```

some compilers give an error, as they cannot fiddle out the correct conversion. This would be to convert `RealValue<double> Product::*` to `RealValue<double> Book::*`, which should be done automatically, and then to instantiate `MemPtr`'s constructor with `RealValue<double>` as `TargetType`.
One way to solve this is to explicitly cast the pointer-to-member:

```
    MemPtr<Book, BaseValue>
        mp2(static_cast<RealValue<double> Book::*>(&Product::price));
```

but that's quite a lot to type. It's actually much easier to move that explicit conversion into the constructor itself and just provide an additional template parameter, as shown in the implementation above. The compiler checks the

conversion anyway, so you will get a compile time error if that conversion is not allowed (e.g. if you try to convert a `RealValue<double> Book::*` to `RealValue<double> Cd::*`.

C++ classes

The `MemPtr`s allow you to access the attribute values of an ordinary C++ object. This helps for one part of the MOP. But what about the attributes themselves? The compiler has the necessary knowledge, but unfortunately there is no standard way to access that knowledge at runtime. So we must provide it and define an interface for it. To allow a smooth integration with our existing `ClassDef`, we provide an `Attribute` iterator as interface. For now, we provide the information about the attributes manually, but in a following article we'll explore the use of a pre-processor for that.
So, for our class `Book` we provide the following functions:

```
class Book : public Product
{
    typedef MemPtr<Book, FinalValue> MemberPtr;
public:
    // as before
    static size_t ownAttribCount()
    {
        return 5;
    }
    static Attribute * ownAttribBegin()
    {
        static Attribute a[]
            = {Attribute("Author", Type::stringT),
               Attribute("Title", Type::stringT),
               Attribute("Publisher", Type::stringT),
               Attribute("Price", Type::doubleT),
               Attribute("Weight", Type::doubleT)
              };
        return a;
    }
    static Attribute * ownAttribEnd()
    {
        return ownAttribBegin() + ownAttribCount();
    }
    static MemberPtr * memberBegin()
    {
        static MemberPtr m[]
            = {MemberPtr(&Book::productNo),
               MemberPtr(&Product::weight),
               MemberPtr(&Book::author),
               MemberPtr(&Book::title),
               MemberPtr(&Book::publisher),
               MemberPtr(&Book::price),
               MemberPtr(&Book::weight)
              };

        return m;
    }
    static MemberPtr * memberEnd()
    {
        return memberBegin() + 7;
    }

private:
```

```
        RealValue<string> author, title, publisher;
        RealValue<double> price, weight;
    };
```

Please note the difference between `ownAttribBegin()` and `memberBegin()`: the first provides information only about the own attributes, while the latter provides access also to base class members. This separation makes sense: while on object level all data members build together one object, on the class level the base class is a different entity and should be available as common base class for the meta object protocol as well. But this separation has consequences: we can't derive a C++ class from a MOP class (but this is definitely no real restriction) and the C++ base class must be also made known to the MOP (but that's useful anyway).
We have no function that provides information about the base classes, as the `baseClass` in `ClassDef` must be made a `ClassDef` instance as well, as noted above. The C++ base classes are not of much use for our application.
With these functions, we can build a `ClassDef` from a C++ class; it's so easy that we can even provide a helper function for that:

```
template <typename CppClass>
ClassDef makeClass(ClassDef const * base, string const & name)
{
    return ClassDef(base, name,
                    CppClass::ownAttribBegin(), CppClass::ownAttribEnd());
}
```

Now, we can build our `ClassDef`s for `Product` and `Book` and create instances from them:

```
ClassDef base(makeClass<DynaProduct>(0, "Product"));
ClassDef book(makeClass<Book>(base, "Book"));
book.newObject();
```

But stop -- though this works, it's not what we want: now, the instances are not genuine C++ objects, but MOP objects, and all access must still go through the MOP.

C++ objects

What we want are real C++ objects that we can also access through the MOP, i.e. through the `Object` interface. The OO way to do that is to derive our `Product` from `Object`, but though other OO languages like Smalltalk do that, I think there's a better, less intrusive option: we provide an adaptor.
From [4] we learn that there are two options for the adaptor pattern: to design the adaptor class as forwarding wrapper class derived only from `Object` and containing a `Product` member or using multiple inheritance and derive the adaptor from `Object` and `Product`. For simplicity, we will use the first option, but real world applications often benefit from the second approach. So we provide a wrapper class `CppObject` that is derived from `Object` and that holds the original C++ object. It implements the interface of `Object` (`getValue()` and `setValue()`) through our `MemPtr`s:

```
template <typename OrigClass>
```

```
class CppObject : public Object
{
    typedef MemPtr<OrigClass, BaseValue> MemberPtr;
public:
    CppObject(ClassDef const * myClass)
     : Object(myClass), myObject(), members(OrigClass::memberBegin())
    {}

    virtual Object * clone() const
    {
        return new CppObject(*this);
    }

    using Object::getValue; // importing getValue(name)
    using Object::setValue; // importing setValue(name)

    virtual Value getValue(size_t idx) const
    {
        return members[idx].value(myObject);
    }

    virtual void setValue(size_t idx, Value const & v)
    {
        BaseValue * p = &(members[idx].value(myObject));
        p->set(v);
    }

private:
    MemberPtr * members;
    OrigClass myObject;
};
```

A useful rule of OO design says that only leaf classes should be concrete, so let's define `Object` as abstract base class and create a new class `DynaObject` that resembles our former `Object` for real MOP class instances (Fig. 7).

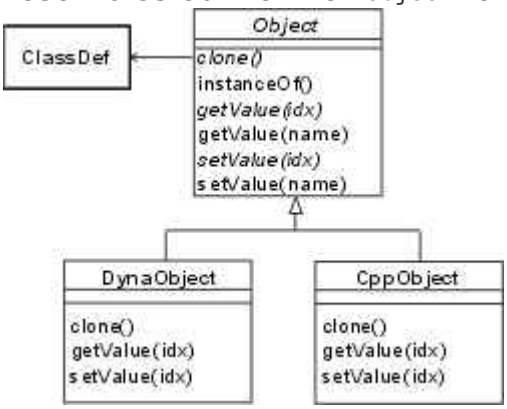

Fig. 7: Object Hierarchy

If we now do a `prod.newObject()`, we still get it wrong: we now get a `DynaObject`, but we want a `CppObject<Product>`. To solve that, we must provide the `ClassDef` with a means to create the correct kind of object, and the simplest way to do that is a factory method: we provide a static creation function in `CppObject<>` and `DynaObject`, give a pointer to that function to the `ClassDef`'s constructor, store it and use that function in `ClassDef::newObject()`:

Creation functions for `DynaObject` and `CppObject`:

```
Object *
DynaObject::newObject(ClassDef const * myClass)
{
```

```
    return new DynaObject(myClass);
}

template <typename OrigClass>
Object *
CppObject<OrigClass>::newObject(ClassDef const * myClass)
{
    return new CppObject(myClass);
}
```

Changes to `ClassDef`:

```
class ClassDef
{
public:
    typedef Object * (*CreateObjFunc)(ClassDef const *);

    template <typename Iterator>
    ClassDef(ClassDef const *, string const &,
             CreateObjFunc objFunc,
             Iterator, Iterator)
     : // ...
       createObj(objFunc)
    {
        // ...
    }

    ClassDef(ClassDef const *, string & const name_,
             CreateObjFunc objFunc)
     : // ...
       createObj(objFunc)
    {
        // ...
    }

    Object * newObject() const
    {
        definitionFix = true;
        return (*createObj)(this);
    }

    // ... as before

private:
    const CreateObjFunc createObj;
    // ... as before
};
```

And a simple change to `makeClass`:

```
template <typename CppClass>
ClassDef makeClass(ClassDef const * base, string const & name)
{
    return ClassDef(base, name,
                    CppObject<CppClass>::newObject,
                    CppClass::ownAttribBegin(),
                    CppClass::ownAttribEnd());
}
```

Now, everything works. Well -- nearly. If we now try to create a `ClassDef` for `Product` with `makeClass` the compiler complains about creating an abstract class:

`makeClass` gives the `ClassDef` constructor a pointer to `CppObject<Product>::newObject()`, and that creates a `Product` instance as part of `CppObject<Product>`. This is easily fixed: just call the `ClassDef` constructor directly with a null-pointer for the creation function, thus prohibiting the creation of a Product instance through the MOP.

Usage

The MOP, as you have it now, allows you to define the C++ classes as MOP classes as before:

```
ClassDef base(0, "Product", 0,
              Product::ownAttribBegin(),
              Product::ownAttribEnd());
ClassDef book(makeClass<Book>(&base, "Book"));
```

You can create instances of them
`book.newObject();`
You can define new classes derived from `Product`

```
ClassDef * tShirt
    = new ClassDef(&base, "T-Shirt",
          DynaObject::newObject);

tShirt->addAttribute(Attribute("Size", Type::stringT));
tShirt->addAttribute(Attribute("Color", Type::stringT));
tShirt->addAttribute(Attribute("Name", Type::stringT));
tShirt->addAttribute(Attribute("Price", Type::doubleT));

classReg.registerClass(tShirt);
```

and manipulate instances of existing classes and new classes through the MOP:
A C++ object:

```
Object * ecpp(book.newObject());

ecpp->setValue(5, RealValue<double>(22.50));
ecpp->setValue(0, RealValue<int>(23456));
ecpp->setValue(2, RealValue<string>("Scott Meyers"));
ecpp->setValue("Title", RealValue<string>("Effective C++"));
ecpp->setValue(6, RealValue<double>(280));
size_t idx;

cout << "ecpp:" << endl;
for (a = book.attribBegin(), idx = 0;
     a != book.attribEnd();
     ++a, ++idx)
{
    cout << a->getName() << ": "
         << ecpp->getValue(idx).asString() << endl;
}

cout << ecpp->getValue("Author").asString() << endl;
```

And a dynamic object:

```
Object * ts(tShirt.newObject());
```

```
ts->setValue(0, RealValue<int>(87654));
ts->setValue(2, RealValue<string>("XXL"));
ts->setValue("Color", RealValue<string>("red"));
ts->setValue("Price", RealValue<double>(25.95));
ts->setValue("Weight", RealValue<double>(387));

for (size_t idx = 0; idx != 4; ++idx)
{
    cout << ts->getValue(idx).asString() << endl;
}
```

C++ Interface

You can't access the instances created by the MOP through the `Product` interface. For instances of C++ classes you can modify the `CppObject<T>` to derive from `T` as we have discussed before, or better, to provide a member function in `CppObject` that returns a pointer to `myObject`. And for instances of MOP classes you can define a wrapper around an `Object` that implements the `Product` interface:

```
class DynaProduct : public Product
{
public:
    DynaProduct(Object const * o) : obj(o) {}
    virtual std::string getName() const
    {
        Value v = obj->getValue("Name");
        return v.get<std::string>();
    }
    virtual double getPrice() const
    {
        Value v = obj->getValue("Price");
        return v.get<double>();
    }

    virtual double getWeight() const
    {
        Value v = obj->getValue("Weight");
        return v.get<double>();
    }

private:
    Object const * const obj;
};
```

And you can't access normal C++ instances of `Book` through the MOP; to solve this, you could add another constructor that adopts the C++ instance by copying (and consequently you should delete the original one to avoid an object that exists twice) or you could modify the wrapper to hold only a pointer to the C++ object and add a member function to return the controlled C++ object on request: Here, we use the first approach:

```
template <typename OrigClass>
CppObject<OrigClass>::CppObject(ClassDef const * myClass,
                                OrigClass const & obj)
 : Object(myClass),
   myObject(obj), // calls the copy-ctor of OrigClass,
                  which must be accessible
```

```
        members(OrigClass::memberBegin())
    {}
```

And now, you can do this:

```
Book b("Bjarne Stroustrup", "The C++ Programming Language",
       "Addison-Wesley", 27.50, 370);
CppObject<Book> mb(*book, b);
Object * ob = &mb;
cout << "C++ object through MOP" << endl;
for (a = ob->instanceOf()->attribBegin(), idx = 0;
     a != ob->instanceOf()->attribEnd();
     ++a, ++idx)
{
    cout << a->getName() << ": "
         << ob->getValue(idx).asString() << endl;
}
```

But in general, it's just important that you can define basic classes in C++ at programming time but allow the user to derive own classes from these base classes at runtime.

Applications

Is such a MOP approach for C++ actually useful? The pure reflection mechanism based on pointers-to-members is quite useful for persistence libraries -- on relational databases or file formats like XML. This was the application when I first used pointers-to-members in C++, which were just the C++ replacement of the old C `offsetOf` macro that is still in widespread use for that purpose.
But I also came across quite a lot of applications where a handful of pre-defined entities provided 98% of the requirements of the users of the system, but the remaining 2% were so different for different users that a common solution for all was not adequate. For these special cases, a full meta-object protocol approach was really a quite simple and elegant solution that provided all the flexibility the users requested.

Two final remarks

Could the same kind of reflection be achieved with `get`/`set` functions instead of pointers-to-members? Perhaps, yes. Some component mechanisms use that approach (e.g. Borland's VCL). In that case, a `get()` and `set()` function for each attribute and a specialized `TypedMemPtr<>` for each attribute for each class is required. That's a lot of work, but with a respective pre-processor or compiler support that's not a point. But it's still much more intrusive to add all the getters and setters; and if they are public, they break encapsulation. Though pointers-to-members allow direct access to the data, that encapsulation leak can be much better controlled.

The second remark relates to `RealValue<>`. Are they really necessary for true reflection purpose? Actually not. In that case, you could remove `BaseTargetType`

from all `MemPtr` templates, return the `TargetType` in `TypedMemPtr`'s getter function, and make the getter function of the `MemPtr` handle a member template function analogous to the getter function of `Value`. In the source code for this article (http://www.vollmann.com/download/mop/index.html) you'll find a sample implementation for that.
Though this seems to be a major advantage for pure reflection applications like persistency libraries, in fact I found it in most cases quite useful to have a base class like `DbValue` for all persistent attributes to provide additional functionality like dirty flags, type conversion specifics, etc.

Coming articles

This article provided reflection and meta-class facilities for data attributes only. A following article will show the application of a MOP for the integration of a scripting language. And that will then allow to extend the MOP with member functions as well.
Another article will look into the capabilities of preprocessors to provide the reflection information.
And yet another article will look at real applications for reflection and meta-classes, like DB libraries.

References

[1] Gregor Kiczales, Jim des Rivières, Daniel G. Bobrow: "The Art of the Metaobject Protocol", MIT Press 1991, ISBN 0-262-61074-4

[2] James O. Coplien: "Advanced C++ Programming Styles And Idioms", Chapters 8-10, Addison-Wesley, 1992, ISBN 0-201-54855-0

[3] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal: "Pattern-Oriented Software Architecture: A System of Patterns", Wiley 1996, ISBN 0-471-95869-7

[4] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: "Design Patterns", Addison-Wesley 1994, ISBN 0-201-63361-2

[5] Michael S. Ball, Stephen D. Clamage: "Pointers-to-Members", C++ Report 11(10):8-12, Nov/Dec 1999

Notes

[1] I will not elaborate on differences between type and class.
[2] The idea of the search machine is to have a batch process that queries all objects about its attribute values and create an own internal index from that information.
[3] actually, a simple array would suffice, as its size is fix and known at compile time. But for future extensions a vector is more flexible.
[4] As `class` is a reserved word in C++, and I don't like identifiers to differ only

in case from others I have chosen `ClassDef` here.

[5] `fromString()` would also be useful, but we omit it here.

[6] In fact, a pointer-to-member is not as easy as an offset. Especially if multiple inheritance comes in, things become more complicated.

[7] It might be necessary to adjust that offset, but the compiler cares for that.