
Subject: Polymorphic XMLizer

Posted by [mdelfede](#) on Wed, 13 Jan 2010 16:52:00 GMT

[View Forum Message](#) <> [Reply to Message](#)

Well, here the polymorphic XMLizer; is able to store/retrieve a polymorphic array of objects on XML. It's based on previous ClassFactory.

It's templated and easy to use; I'll post here the code, will post in Bazaar on next days.

ClassFactory.h

```
#ifndef _ClassFactory_h_
#define _ClassFactory_h_

#include <Core/Core.h>

NAMESPACE_UPP;

template<class T> class WithFactory
{
private:
    typedef One<T> (*CreateFunc)();
    typedef VectorMap<String, CreateFunc>mapType;
    static mapType &classMap() { static mapType cMap; return cMap; }
    static VectorMap<String, String> &typeMap() { static VectorMap<String, String> tMap; return
tMap; }
    template<class D> static One<T> __Create(void) { return One<T>((T *)new D); }
public:

    template<class D> static void Register(const String &name)
    {
        classMap().Add(name, __Create<D>);
        typeMap().Add(typeid(D).name(), name);
    }
    static One<T> Create(const String &className) { return classMap().Get(className()); }
    static T *CreatePtr(String const &className) { return classMap().Get(className()).Detach(); }
    static Vector<String> const &Classes(void) { return classMap().GetKeys(); }
    String const &IsA(void) { return typeMap().Get(typeid(*this).name()); }
    virtual ~WithFactory() {}
};

#define REGISTERCLASS(type) \
    INITBLOCK { \
        type::Register<type>(#type); \
    }

END_UPP_NAMESPACE;

#endif
```

PolyXML.h

```
#ifndef _PolyXML_h_  
#define _PolyXML_h_
```

```
#include "ClassFactory.h"
```

```
NAMESPACE_UPP;
```

```
template<class T> class WithPolyXML : public WithFactory<T>  
{  
public:  
    // Xmlizer  
    virtual void Xmlize(XmlIO xml) {};  
};
```

```
template<class T> class PolyXMLArray : public Array<WithPolyXML<T> >  
{  
public:  
    // Xmlizer  
    void Xmlize(XmlIO xml);  
    void Add(const T &data) { Array<WithPolyXML<T> >::Add(data); }  
    void Add(T *data) { Array<WithPolyXML<T> >::Add(data); }  
};
```

```
template<class T> void PolyXMLArray<T>::Xmlize(XmlIO xml)  
{  
    if(xml.IsStoring())  
    {  
        for(int i = 0; i < PolyXMLArray::GetCount(); i++)  
        {  
            WithPolyXML<T> &data = PolyXMLArray::operator[](i);  
            String tag = data.IsA();  
            data.Xmlize(xml.Add(tag));  
        }  
    }  
    else  
    {  
        PolyXMLArray::Clear();  
        for(int i = 0; i < xml->GetCount(); i++)  
        {  
            if(xml->Node(i).IsTag())  
            {  
                String tag = xml->Node(i).GetTag();  
                T *data = T::CreatePtr(tag);  
                if(data)
```

```

    {
        data->Xmlize(xml.At(i));
        Add(data);
    }
}
}
}
}
}

```

```
END_UPP_NAMESPACE;
```

```
#endif
```

Sample usage (with testing); it creates a polymorphic array with 2 elements (one of base class, one of a derived one), streams to an XML string and load back from it, printing its contents before and after streaming :

```
#include "PolyXML.h"
```

```
using namespace Upp;
```

```

class Base : public WithPolyXML<Base>
{
public:
    String BaseData;
    void Xmlize(XmlIO xml) { xml("BaseData", BaseData); }
};

```

```

class Derived : public Base
{
public:
    String DerivedData;
    void Xmlize(XmlIO xml) { Base::Xmlize(xml); xml("DerivedData", DerivedData); }
};

```

```

REGISTERCLASS(Base);
REGISTERCLASS(Derived);

```

```

CONSOLE_APP_MAIN
{
    PolyXMLArray<Base> polyArray;

```

```

    Base *b = new Base;
    b->BaseData = "Sample data in base class";
    polyArray.Add(b);

```

```

Derived *d = new Derived;
d->BaseData = "Sample data in derived class";
d->DerivedData = "Another sample data in derived class";
polyArray.Add(d);

Cerr() << "\nArray content before streaming out: " << polyArray.GetCount() << " classes:\n";
for(int i = 0; i < polyArray.GetCount(); i++)
{
    Cerr() << " class #" << i << " is a " << polyArray[i].IsA() << "\n";
    if(polyArray[i].IsA() == "Base")
        Cerr() << "   BaseData   = " << ((Base &)polyArray[i]).BaseData << "\n";
    else
    {
        Cerr() << "   BaseData   = " << ((Derived &)polyArray[i]).BaseData << "\n";
        Cerr() << "   DerivedData = " << ((Derived &)polyArray[i]).DerivedData << "\n";
    }
}

String s = StoreAsXML(polyArray, "PolyXMLTest");

Cerr() << "\nStreamed XML : \n\n" << s ;

polyArray.Clear();
LoadFromXML(polyArray, s);

Cerr() << "\nArray content after streaming in : " << polyArray.GetCount() << " classes:\n";
for(int i = 0; i < polyArray.GetCount(); i++)
{
    Cerr() << " class #" << i << " is a " << polyArray[i].IsA() << "\n";
    if(polyArray[i].IsA() == "Base")
        Cerr() << "   BaseData   = " << ((Base &)polyArray[i]).BaseData << "\n";
    else
    {
        Cerr() << "   BaseData   = " << ((Derived &)polyArray[i]).BaseData << "\n";
        Cerr() << "   DerivedData = " << ((Derived &)polyArray[i]).DerivedData << "\n";
    }
}
}

```

The only noticeable stuff is that you MUST define Xmlize function for each class of hierarchy AND you MUST call direct base class Xmlize() of each derived class.

Sample output :

Array content before streaming out: 2 classes:
class #0 is a 'Base'

```
BaseData  = 'Sample data in base class'
class #1 is a 'Derived'
BaseData  = 'Sample data in derived class'
DerivedData = 'Another sample data in derived class'
```

Streamed XML :

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<!DOCTYPE PolyXMLTest>
<PolyXMLTest>
  <Base>
    <BaseData>Sample data in base class</BaseData>
  </Base>
  <Derived>
    <BaseData>Sample data in derived class</BaseData>
    <DerivedData>Another sample data in derived class</DerivedData>
  </Derived>
</PolyXMLTest>
```

Array content after streaming in : 2 classes:

```
class #0 is a 'Base'
BaseData  = 'Sample data in base class'
class #1 is a 'Derived'
BaseData  = 'Sample data in derived class'
DerivedData = 'Another sample data in derived class'
```

Ciao

Max
