METASPEX

Metaspex Technical Insights

# III - Manage Objects & Point at Them

# Introduction

Metaspex manages objects which participate in documents. Objects are interrelated through **own** relationships within documents, and can point at each other through **link**s which resemble HTML URLs.

Each object is allocated on the heap (in RAM), and persistence (save and retrieval) is automatically managed–save through serialization, and retrieval through deserialization.

The purpose of this document is to explain how to write types' constructors, how to allocate objects, how to point at them through Metaspex smart pointers, and how to set relationships from one object to another.

At the end of the document we'll explain how objects are automatically and immediately deallocated when they are not in use any longer.

# Constructors and the make function

Let’s take the address type we defined earlier:

```
#include <string>

#include "hx2a/element.hpp"
#include "hx2a/slot.hpp"

using namespace std;
using namespace hx2a;

namespace my_ontology {
   class address: public element<>
   {
      HX2A_ELEMENT(address, "addr", element, (_number, _street, _city, _state,
_zipcode));
      slot<unsigned int, "num"> _number;
      slot<string, "strt"> _street;
      slot<string, "cty"> _city;
      slot<string, "st"> _state;
      slot<string, "zip"> _zipcode;
   };
}
```

And add a constructor for that type. With Metaspex it’ll take a specific form:

```
address(unsigned int number, string street, string city, string state, string
zipcode):
   _number(*this, number),
   _street(*this, street),
   _city(*this, city),
   _state(*this, state),
   _zipcode(*this, zipcode)
{
}
```

Note the usage of *this as the first argument of each **slot** constructor. This is mandatory. It is optionally followed by the initialization value. If that value is not present, the **slot** is default-initialized.

We wrote a simple signature, using no string const reference and using no move, to concentrate on Metaspex itself rather than C++ mechanics. All these can be used if you wish to.

So, adding access specifiers, the address type now looks like:

```
  class address: public element<>
   {
      HX2A_ELEMENT(address, "addr", element, (_number, _street, _city, _state,
_zipcode));
   public:
      address(unsigned int number, string street, string city, string state,
string zipcode):
      _number(*this, number),
      _street(*this, street),
      _city(*this, city),
      _state(*this, state),
      _zipcode(*this, zipcode)
      {
      }
   private:
      slot<unsigned int, "num"> _number;
      slot<string, "strt"> _street;
      slot<string, "cty"> _city;
      slot<string, "st"> _state;
      slot<string, "zip"> _zipcode;
   };
```

How do we allocate an address object? That's the purpose of Metaspex's **make** function:

```
make<address>(42, "Metaspex Rd", "Boston", "MA", "55555");
```

The arguments of the **make** template function are passed directly to the type's constructor. 42 becomes the number in the street, "Metaspex Rd" becomes the street, etc.

In the case of a **root** or an **entity**, the **make** function takes an additional argument, coming first: the database connector.

Using a survey type taking a string name in its constructor, the code creating a survey document which persists in database "hx2a" would be:

```
#include "hx2a/db/connector.hpp"

#include "survey.hpp"

void f(){
   db::connector c("mydb");
   make<survey>(c, "my survey");
}
```

The characteristics of database "mydb" are specified in the Metaspex configuration file, read by the application at startup. Here we just need to specify a logical name ("mydb") to indicate that the newly-created document will persist in that database.

No database API, no query language, and no serialization function is needed for the survey to persist in the "mydb" database. No document identifier needs to be specified; a document identifier (a 128-bit random UUID) is automatically allocated. The code above is complete.

Connectors automatically cache and reuse database connections, so there is no need to manage this explicitly.

To remove an entire document, calling the **unpublish** function on the **root** object is the only operation necessary. Here again, no database API or query language needed. Calling **unpublish** will cause all the external **link**s pointing at all the objects inside the document to be broken by referential integrity. It will not cause all the objects to be destroyed on the heap. It is a logical operation.

# Rfrs and ptrs

Metaspex enables the management of relationships between objects, but also of smart pointers pointing at them. “Smart pointer” is to be understood as using pointing techniques that go beyond regular pointers and perform some kind of additional task. In the case of Metaspex one of the additional tasks is to allow automatic object deallocation. Objects are deallocated immediately when they are not used any longer.

Metaspex has two kinds of smart pointers, so-called **rfr**- and **ptr**-based ones. The difference between these two is simple: **rfr**s cannot be null, while **ptr**s can be null. It means that when a function receives an **rfr**, it does not need to check whether it is null or not before executing some kind of operation on the pointed object.

The **make** function we presented above naturally returns an **rfr**. It allocates a new object, and the **rfr** returned points at it.

Both **rfr** and **ptr** are template types, so an **rfr** to an address is **rfr**<address>, while a **ptr** to an address is **ptr**<address>. Although it is not mandatory, we often use internal aliases for **rfr**s and **ptr**s, such as:

```
using address_p = ptr<address>;
using address_r = rfr<address>;
```

Metaspex's Foundation Ontology uses these abundantly as shorthands.

Similarly, for a **root**, **rfr**<survey> and **ptr**<survey> would respectively be the **rfr** and the **ptr** to a survey.

To test if a **ptr** is null, you can use:

```
void f(address_p addr){
   if (addr == nullptr){
      // Some behavior.
   }
}
```

Since **rfr**s and **ptr**s both implement automatic reference counting, we tend to use this more sophisticated style:

```
void f(const address_p& addr){
   if (addr == nullptr){
      // Some behavior.
   }
}
```

It helps some compilers avoid unnecessary reference counting handling. While passing **rfr**s and **ptr**s as const references is not mandatory, it might be helpful in order to save every bit of performance.

An advantage of **rfr**s is that if a function receives an **rfr**, it can pass it along to other functions and they also benefit from the property that **rfr**s cannot be null. They do not have to test the smart pointers to check that they are not null:

```
void f(const address_r& addr){
   // No need to test if addr is null, we know it's not.
}

void g(const address_r& addr){
   f(addr); // Passing along the rfr.
}
```

With Metaspex, you do not use regular C++ pointers and references on Metaspex objects. These do not manage objects' lifespan, and are meant for low-level code where saving processor cycles makes sense. At the specification level targeted by Metaspex, **rfr**s and **ptr**s are the only ways to point at objects other than through relationships. (Relationships are discussed in the earlier *Ontology* paper, and also later in this paper.) With **rfr**s and **ptr**s you do not run the risk of pointing at a deleted object, as they ensure that objects are destroyed as soon as they are no longer being used. If a piece of code handles an **rfr** or a **ptr** to an object, you can be sure the object pointed at has not been deleted.

# Conversion from rfr to ptr and vice versa

An **rfr** will convert (we could say "decay") into a **ptr** implicitly. Converting a **ptr** to an **rfr** requires more effort.

So the following code is perfectly correct:

```
rfr<address> f(){
   return make<address>(42, "Metaspex Rd", "Boston", "MA", "55555");
}

ptr<address> g(){
   return f(); // rfr<address> "decays" into ptr<address>.
}
```

An **rfr** possesses the "->" operator, a **ptr** does not. So, to access a member of the pointed type, an **rfr** will be necessary. It means that if you have a **ptr**, a conversion into an **rfr** will have to be performed.

Here is one way to do it:

```
rfr<address> g(const ptr<address>& addr){
   return *addr; // ptr<address> is "promoted" into a rfr.
}
```

The "*" operator turns a **ptr** into an **rfr**. Of course, this operation should only be performed in the case where you are completely sure the **ptr** is non-null. In the case where you need to throw an exception if the **ptr** is null, another way to perform the conversion is possible:

```
struct my_exception{};

rfr<address> g(const ptr<address>& addr){
   return addr.or_throw<my_exception>();
}
```

Note that **or_throw** will check that the **ptr** is non null. If it is not, it will upgrade the **ptr** into an **rfr**; otherwise, it will build an exception object of the type specified, with the arguments given, and will throw it. Consider using one of the ready-made exceptions given by Metaspex to have graceful exception replies.

# Rfrs, ptrs and relationships

While **rfr**s and **ptr**s can "sit" in the stack, holding objects and preventing them from being automatically destroyed, we also have relationships between objects, as specified in the ontology.

All relationships of the **own** and **link** families can be null. In some sense, they are **ptr**s. This holds true even if these relationships are strong. A **strong_own** or a **strong_link** can be null. However, it does not mean that these abstractions sit on top of **ptr**s. **Own** relationships do, while **link** relationships don't. Referential integrity guarantees that **link** relationships never point at destroyed objects. It is a theorem, and it is unique to Metaspex.

It can be understood easily through *ad absurdum* reasoning. For a link to point at a destroyed object, it would have to mean the target object did not have a single **rfr** or **ptr** pointing at it. There are two cases: either the target is a **root**, or it is an **anchor**.

- If it is a **root**, it means that the corresponding document has been entirely destroyed, and this is only possible if the document was explicitly removed, by calling the **unpublish** function on its **root** object. Removing a document cuts all the **link**s pointing at it, so this is impossible.
- If it is an **anchor**, it means that no **own** relationship points at it. If there was one, it contains a **ptr**, and that **ptr** would have prevented the destruction of the object. If no **own** relationship points at the object, no **link** can point at it either, because of referential integrity. This is not possible either.

So we see that a byproduct of the high-level referential integrity process is that **link**s never point at a destroyed object, in spite of the fact that they do not contain any smart pointers (**rfr** or **ptr**).

Objects are automatically deallocated as soon as they are not used any longer, which means as soon as they are not pointed at by a smart pointer (**rfr** or **ptr**) and when they do not participate in a document.

Now, assigning an **own** or **link** relationship is simple: you just need to assign them an **rfr** or a **ptr**. These relationships also convert implicitly into **rfr** and **ptr**.

When accessing the target object of a **link** (just by writing “o->_l”, for instance), if the target is not yet “resolved,” meaning that the target document has not been acquired previously by the application, Metaspex will automatically retrieve the document, put it in the application cache, and assign the **link** its expected value. No database API or query language is involved, whether the target of the **link** is in the same database bucket as the origin object, a different bucket, a different database instance, a different database product, or even in a different data center if necessary.

When composite relationships are involved, just consult the reference guide for the APIs. They handle **rfr**s and **ptr**s, depending on the situation.

# Conclusion

Metaspex offers unique ways to point at objects on the heap, while object-to-object relationships are engineered in a way that makes them live in harmony with Metaspex smart pointers.

Relationships are subject to automatic referential integrity, which performs automatic modifications of documents to keep their integrity. On the other hand, smart pointers allow immediate destruction of objects when they are not used any longer. This means that referential integrity and information destruction in memory are two distinct processes, not tied together, yet working together. This is a substantial difference from primitive referential integrity as implemented in RDBMS, where referential integrity is directly tied to information lifecycle on disk. Here, Metaspex distinguishes two useful levels of abstraction: referential integrity and automatic object destruction.

Code that uses Metaspex contains no explicit object destruction, database API calls, or query languages, and this establishes Metaspex user code as a high-level specification, not low-level software full of boilerplate.