1 28-05-2020

I have recently been building a API around the redland libraries (raptor, rasqal and redland/librdf). The idea is to wrap the underlying C classes in C++ classes, making use of constructors and destructors to enable support for RAII. Instead of doing things like

```
librdf_world* world = librdf_new_world();
librdf_node* node = librdf_new_node_from_uri_string(world, "https://uri.com");
librdf_free_node(node);
librdf_free_world(world);
```

we do things like

```
// world argument is handled internally.
LibrdfNode node = LibrdfNode::fromUriString("https://uri.com");
```

The intention is for these two pieces of code to be equivalent - memory in the second is handled automatically. This is extra work now, but if done correctly, it means that new developers do not have to deal with the memory related issues that I am dealing with.

The mechanism I am using for wrapping these classes involves using a unique pointer to the C resource that is constructed either from a pointer to the C object or from the arguments required to build the C object. For example:

```
...
// deleter for a node wrapper. Will be called automatically when
// LibrdfNode obj goes out of scope.
void LibrdfNode::deleter::operator()(librdf_node *node) {
    librdf_free_node(node); // the redland call to delete a node
}

\\ create from a librdf_node
LibrdfNode::LibrdfNode(librdf_node *node)
        : node_(std::unique_ptr<librdf_node, deleter>(node)) {
    uri_ = LibrdfUri(librdf_node_get_uri(node_.get()));
}

\\ from a string to a blank node (actually a
\\ static method for creating blank LibrdfNode)
LibrdfNode LibrdfNode::fromBlank(const std::string &blank) {
    return LibrdfNode(
        librdf_new_node_from_blank_identifier( // librdf command
            World::getWorld(), // static singleton method for getting world
            (const unsigned char *) blank.c_str())
        );
```

```
    }
...
```

I have been developing this API along side the code for libsemsim but there are complications that I thought I had resolved, but have resurfaced when these objects interact. One such complication is that the C function responsible for freeing a librdf_node* will implicitely call the C function responsible for freeing a librdf_uri* (that is contained within). Therefore, when the c++ scope ends C++ tries to free both the LibrdfNode and the LibrdfUri, but the calls to the C library also try to free both resoures. So we have a double freeing problem.

To resolve this I planned on copying the code for the destructors from the C libraries and making the required modifications such that each object is only responsible for its own destruction. This is possible, but requires access to some structs from the C libraries that are private. Otherwise we get "access to incomplete type" errors. The first thing I tied was to copy the structs I need over from C library so I have a local copy, but this causes a namespace clash. Plus its quite an inelegant solution. An alternative is to develop the wrapper along side the redland project rather the libsemsim project. So I would copy all the wrapper classes over to the project where I'm building the redland api and develop them there. This way I'll have access to the private structs. This would more or less be a copy and paste job if CMake scripts were available for the redland libraries. There not so I would have to develop them first.

So unless I'm misguided, which is certainly a possibility, the order of what I need to do it this:

- develop cmake script for raptor, rasqal and librdf/redland libraries (for linux only, cross platform later).
- Either pull these into the libsemsim superbuild or copy the Redland wrapper api over to where I'm building redland libraries
- Create custom deleter functions for the C objects and ensure valgrind is happy.
- Continue integrating the new wrapper backend into the libsemsim frontend.