Registry Package Rationale

Brett Viren

March 16, 2009

1 Name and Acronym

Name: `Registry`

2 Coordinator

Brett Viren (bv@bnl.gov)

3 Purpose

The purpose of a `Registry` is to dynamically store heterogeneous values which are accessed by name in the form of a key of type string. This is similar to "alists" in Lisp or "associative arrays" in Perl. This can be used to encapsulate configuration data or provide a uniform interface to some data provider.

4 Concepts

A `Registry` has the following simple concepts:

- Key. The *key* names a value in the context of the `Registry`'s map.
- Value. The *value* is some data of arbitrary type.
- Map. The *map* is the internal data structure which allows a value to be looked up by a key.
- Locks. One can *lock* either values or keys. If values are locked then one can not change already existing values. If keys are locked, one can not add any new key/value pairs. These locks are independent from each other.

5 Clients

The initial client for `Registry` is AlgConfig, the Algorithm configuration class.

6 Services Provided

Mapping of strings to heterogeneously typed values in a type safe manner. Encapsulated storage of these key/value pairs. ROOT I/O, assuming all stored values can be sent through ROOT I/O.

7 Services Required

Currently, `Registry` requires no services from other MINOS classes. It does need templates and RTTI (it calles `dynamic_cast()`). It uses ROOT I/O and depends on ROOT mechanisms for this.

8 Input Data

Key/value pairs are given to a `Registry` for storage. The key is passed by value and a copy is stored. The value is passed by reference and a copy is stored. This copied value will be destroyed when the `Registry` is destroyed, if another value is set with its key or if the `Clear()` method is called.

9 Output Data

Values can retrieved by passing keys to the correct templated `Get()` methods. The value can be returend by value, reference or pointer.

10 Error handling

The `Get()` methods return a bool. A false means that the lookup failed and a message is printed. (**FIXME: currently to stderr, doesn't yet use MsgService**). The `Set()` methods also return bool. A false means that performing the set would violate the current locks.

11 State Data

In a sense, the whole purpose of the `Registry` is to hold state data. The state of the registry is kept with a map which allows values to be looked up by a key. This happens to be implemented with an STL map.

12 Dependencies on Other Packages

ROOT I/O. This is not a fundamental dependency and could be easily removed.

13 List of Classes

The Registry package consists of the following classes:

- `Registry` which defines the only user accessable interface.
- `RegistryItem` which is used as a generic and untemplated holder of a Value.
- `RegistryItemXxx` which derives from `RegistryItem` and is the templated holder of a Value.
- `Registry::RegistryKey` which encapsulates a very simple iterator that lets one iterate over all the keys in the Registry.

14 Implicit Assumptions and Consequences

None, as it should be.

15 Design Patterns Used

None, unless `Registry` itself is considered a Design Pattern. It is more likely an Idiom.

16 Requirements and Use Cases

Requirements:

- Encapsulate key/value pairs.
- Store heterogeneous values in a type safe manner.
- Own the values.
- Provide for I/O.
- Allow setting and getting of values.
- Allow iteration over key/value pairs.
- Allow locks to disallow modifying existing values or adding new key/value pairs.

Use cases:

- Algorithm config.
- In NEUGEN (**FIXME: I think Hugh mentioned something about this?**).

17 UML Diagrams

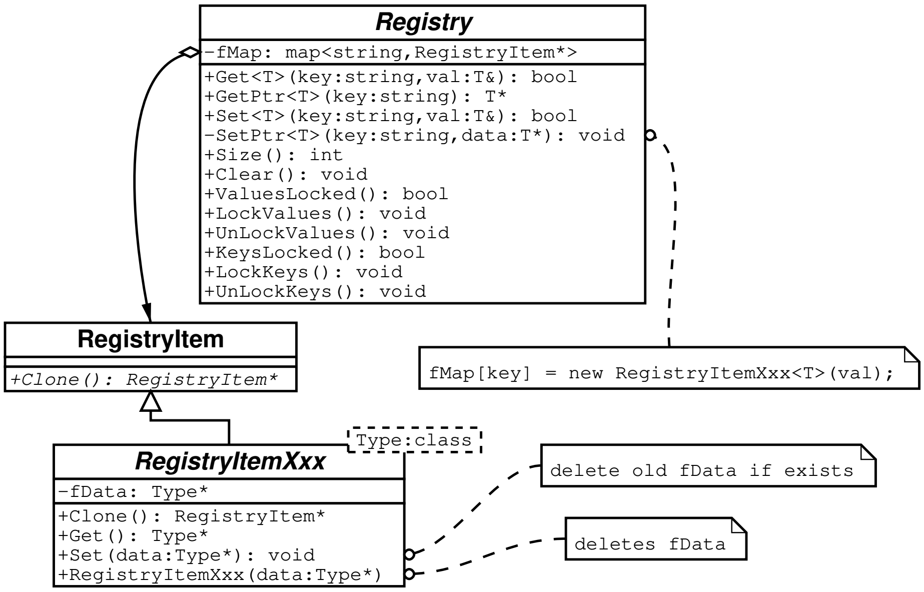

Figure 1: UML diagram showing `Registry` and related classes.

See Fig. 1.

18 Example Usage

```
#include <Registry/Registry>
Registry r;
double d = 6.9;
int i = 69;
char c = ’c’;
char* s = "Blah blah blah!";

r.Set("my double", d);
r.Set("my integer", i);
r.Set("my char", c);
r.Set<char*>("my string", s);

double d = 0;
if (r.Get("my double",d)) cerr << "my double is; ‘" << d << "’\n";
else cerr << "Failed to Get ‘my double’\n";
```

```
int i = 0;
if (r.Get("my integer",i)) cerr << "my integer is; `" << i << "'\n";
else cerr << "Failed to Get `my integer'\n";

char c = 0;
if (r.Get("my char",c)) cerr << "my char is; `" << c << "'\n";
else cerr << "Failed to Get `my char'\n";

char* cptr = 0;
if (r.Get("my string", cptr)) cerr << "my string is; `" << cptr << "'\n";
else cerr << "Failed to Get `my string'\n";

r.LockValues(); // Disallow existing values to be modified
r.LockKeys(); // Disallow addition key/value pairs to be added
```

19 Implimentation Details

`Registry` holds each value using a templated `RegistryItemXxx`. You gain access to these templates by having:

```
#include <Registry/Registry>
```

in your code. To store a type of value other than one of the base C++ types, one must create code to instantiate the necessary templates. In particular, the methods ROOT adds with ClassDef (ClassDefT for templates) must be created.

Assuming one wants to store a TObjString and a user defined class which derives from TObject, say, the class MyClass, then one must create a LinkDef.h file like:

```
#ifdef __CINT__
#pragma link off all globals;
#pragma link off all classes;
#pragma link off all functions;

#pragma link C++ class TObjString;
#pragma link C++ class RegistryItemXxx<TObjString>-;
#pragma link C++ class MyClass;
#pragma link C++ class RegistryItemXxx<MyClass>-;
#endif
```

This will make use of the templated Streamer for the `RegistryItemXxx<>` classes. This will only work if there is a

```
TBuffer& operator>>(Class*&)
```

defined for class Class (and the coressponding `<<` operator). This LinkDef.h file must then be run through rootcint like:

```
rootcint -f MyStuff.cxx -c -I/path/to/include Registry/Registry LinkDef.h
```

and the resulting MyStuff.cxx compiled and linked in to the executable.

20 Initialization and Reset

When a new `Registry` is created the default is to have Values locked and Keys unlocked. This allows one to add new Key/Value pairs, but not to change existing ones. After creation, one can use the `{Lock,UnLock}{Values,Keys}` methods to change this.

When a value is given to a `Registry` a copy is made. Copies will be destroyed either when the `Registry` is destroyed or if a new value is saved with a previously used Key. One can call the `Clear()` method to delete all values and clear the internal map.