N-ary Tree Node

Brett Viren

April 30, 2024

Contents

1 Overview **1**

2 Synopsis **1**

3 Description **2**

4 Notification **2**

5 Facade **3**

1 Overview

WCT provides a N-ary tree data structure in the form of a single class `WireCell::NaryTree::Node`.

2 Synopsis

A synopsis of the most important parts of the API is here:

```
// in WireCellUtil/NaryTree.h
namespace WireCell::NaryTree {
    template<class Value>
    class Node {

        Value value;
        Node* parent;

        using owned_ptr = std::unique_ptr<Node>;
        using nursery_type = std::list<owned_ptr>;

        Node();
        Node(const Value& val);
        Node(Value&& val);

        Node* insert(const Value& val);
        Node* insert(Value&& val);
        Node* insert(owned_ptr node);

        owned_ptr remove(const Node* node);

        nursery_type& children();
```

```
        auto child_values();
        auto child_nodes();
    };
}
```

3 Description

A `Node` has the following parts:

value of an application-determined type given by the `Value` template parameter.

parent a pointer to another **node** that has this **node** as a child or `nullptr` if this **node** is free.

children a collection of child **nodes** that this **node** owns.

A `Node` may be constructed empty, relying on a `Value` to provide a default constructor for the **value**. Alternatively, a `Node` may be constructed either by copying or moving a **value**.

A **child** `Node` may be created by **inserting** its **value** into an existing **parent** `Node`. The child will have its `.parent` value pointing to the **parent** node. A `Node` that has no **parent** is not itself a child and is considered a *root node*. If a `Node` is a child of a parent and it is inserted into another parent `Node` it will be removed from its original parent. When a child `Node` is removed from a parent the `remove()` method returns a `std::unique_ptr<Node>` representing that the parent relinquishes ownership of the child to the caller.

The collection of children of a `Node` may be accessed with its `children()` method. Their are held by `std::unique_ptr<Node>` and thus care must be taken when working directly with the children collection. The `child_values()` and `child_nodes()` provide an iterator range to access the child **values** or to acess the children as simple `Node*`, respectively.

4 Notification

The `Value` of a `Node` will be informed of major state changes in the `Node` that is holding the `Value` if the `Value` defines a method matching this signature:

```
void Value::notify(Node<Value>* node, Action action);
```

The `action` is an enum determines the state change:

constructed the node was just constructed.

inserted the node is a child that was just inserted to a parent.

removing the node is a child that will next be removed from its parent.

The `Value` type may wish to propagate notifications up the parentage. For example, to freshen or invalidate caches held by a parent and that span child nodes.

The `NaryTree::Notified<Value>` class defined in `NaryTreeNotified.h` provides a base class for a `Value` type (CRTP) that helps dispatch an action by hits value to a matching method call. A `Value` type may inherit from this type and define zero or more of these methods:

```
virtual void on_construct(node_type* node);
virtual bool on_insert(const std::vector<node_type*>& path);
virtual bool on_remove(const std::vector<node_type*>& path);
```

The `path` argument gives the tree path to the node that triggered the notification.

5 Facade

The `NaryTree::` name space provides additional classes to facilitate building a tree of nodes with some heterogeneity of `Value` types. It does this through the idea of a "facade" which represents some code that is "attached" to a node. Code that is both external or internal to **facade**, **value** and **node** can all inter-navigate between these objects. A facade instance is held by `std::unique_ptr`. This means the lifetime of the facade is bound to that of the node (and the value). It also means the facade may be of a polymorphic type and this is what allows for heterogeneity of type across the tree nodes. Note, these all follow the CRTP.

- `Facade<Value>` is a `Notified<Value>` and will use the `on_construct()` notification to collect the node pointer and make it available to external code or a sub class.
- `Faced<Value>` functionality to hold a facade instance. A `Value` that inherits from this type enables a tree nodes to hold a variety of unique facade types inheriting from `Facade<Value>`. A `Faced` is itself a `Facade` which, in principle, allows for chaining of types.
- `FacadeParent<Child,Value>` base is a `Facade<Value>` that provides code to deal with its node's child nodes via their `Facade<Child>` types. A tree using this facade can be seen as homogeneous across each layer and heterogeneous down the layers.

Putting this together, here is an example excerpted from `doctest-pointtree-facade.cxx` with `L0` providing the type of facade for the root layer node, `L1` the middle layer and `L2` the leaf layer.

```
struct Value : public Faced<Value> { ... };
struct L2 : public Facade<Value> { ... };
struct L1 : public FacadeParent<L2, Value> { ... };
struct L0 : public FacadeParent<L1, Value> { ... };

auto root = build_tree();
auto* l0 = root->facade<L0>();
auto* l1 = l0->children()[0];    // an L1 facade
auto* l2 = l1->children()[0];    // an L2 facade
```