xproperty

Release

Nov 24, 2017

INSTALLATION

C++ properties and observer pattern

`xproperty` is a C++ library providing traitlets-style properties.

`xproperty` provides an implementation of the observer patterns relying on C++ template and preprocessor metaprogramming techniques.

Properies of observed objects have no additional memory footprint than the value they hold. The assignment of a new value is simply replaced at compiled time by

- the call to the validator for that property
- the actual underlying assigment
- the call to the observor for that property.

We also provide the implementation of an `xobserved` class whose static validator and observer are bound to a dynamic unordered map of callbacks that can be registered dynamically.

`xproperty` requires a modern C++ compiler supporting C++14. The following C++ compilers are supported:

- On Windows platforms, Visual C++ 2015 Update 2, or more recent
- On Unix platforms, gcc 4.9 or a recent version of Clang

CHAPTER 1

Licensing

We use a shared copyright model that enables all contributors to maintain the copyright on their contributions.

This software is licensed under the BSD-3-Clause license. See the LICENSE file for details.

1.1 Installation

xproperty is a header-only library. We provide a package for the conda package manager.

```
conda install -c conda-forge xproperty
```

Or you can directly install it from the sources:

```
cmake -D CMAKE_INSTALL_PREFIX=your_install_prefix
make install
```

1.2 Compiler workarounds

This page tracks the workarounds for the various compiler issues that we encountered in the development. This is mostly of interest for developers interested in contributing to xtensor.

1.2.1 Visual Studio and internal classes

In `xproperty` properties are internal classes that implement a CRTP pattern. Unlike most CRTP patterns implemented in QuantStack projects, the internal `typedef` to the derived class is not called `derived_type` but `xp_derived_type`. The reason for this, is that the way MSVC resolves typedefs of internal classes is bugged: if the owner itself has a `derived_type` typedef, its value is going to be used instead.

In other words, the outer class typedefs mask the inner class typedefs.

1.3 Usage

1.3.1 Basic Usage

- Declaring an observed object `Foo` with two properties named `bar` and `baz` of type *double*.
- Registering a validator, executed prior to assignment, which can potentially coerce the proposed value.
- Registering a notifier, executed after the assignement.

```
#include <iostream>
#include <stdexcept>

#include "xproperty/xobserved.hpp"

struct Foo : public xp::xobserved<Foo>
{
    XPROPERTY(double, Foo, bar);
    XPROPERTY(double, Foo, baz);
};
```

Registering an observer and a validator

```
Foo foo;

XOBSERVE(foo, bar, [](const Foo& f) {
    std::cout << "Observer: New value of bar: " << f.bar << std::endl;
});

XVALIDATE(foo, bar, [](Foo&, double proposal) {
    std::cout << "Validator: Proposal: " << proposal << std::endl;
    if (proposal < 0)
    {
        throw std::runtime_error("Only non-negative values are valid.");
    }
    return proposal;
});
```

Testing the validated and observed properties

```
foo.bar = 1.0;                          // Assigning a valid value
                                        // The notifier prints "Observer: New value␣
↪of bar: 1"
std::cout << foo.bar << std::endl;      // Outputs 1.0

try
{
    foo.bar = -1.0;                     // Assigning an invalid value
}
catch (...)
{
    std::cout << foo.bar << std::endl;  // Still outputs 1.0
}
```

Shortcuts to link properties of observed objects

```
// Create two observed objects
Foo source, target;
```

```
source.bar = 1.0;

// Link `source.bar` and `target.baz`
XDLINK(source, bar, target, baz);

source.bar = 2.0;
std::cout << target.baz << std::endl;     // Outputs 2.0
```

1.3.2 Advanced Usage: Using *XPROPERTY* without *xobserved*

The standard usage of `XPROPERTY`, with the CRTP inheritance from `xobserved` is not optimal if you known at build time what validator and notifiers should be called.

Indeed, the `xobserved`, which allows the dynamic registration of validators and notifiers stores those in unordered maps, which

- increases the memory footprint of the observed object
- results in the `O(1)` lookup into the hash map.

Instead, you can use `XPROPERTY` alone, to remove that overhead.

```
#include <iostream>
#include <stdexcept>

#include "xproperty/xproperty.hpp"

struct Foo
{
    XPROPERTY(double, Foo, bar);

    MAKE_OBSERVED();

    XVALIDATE_STATIC(double, Foo, bar, proposal)
    {
        std::cout << "Validator: Proposal: " << proposal << std::endl;
        if (proposal < 0)
        {
            throw std::runtime_error("Only non-negative values are valid.");
        }
        return proposal;
    });

    XOBSERVE_STATIC(double, Foo, bar)
    {
        std::cout << "Observer: New value of bar: " << f.bar << std::endl;
    });
}
```

In this case, the assignment of the `Foo::bar` property will simply be replaced by the calls to the validator and notifier at build time without any overhead in memory footprint.

- Unlike dynamic validators and notifiers, static validators and notifiers are methods of the observed objects.
- One cannot mix the static and dynamic validators and observers