PLAXIS

essential for geotechnical professionals

smartref - Start using smart references in your code today

A modern header-only zero-overhead library for creating smart references

Erik Valkering (eejv@plaxis.com)

Plaxis bv, Delft, The Netherlands

Abstract

smartref is a *pure-library* solution for creating smart reference classes using C++11/14/17, and comprises two parts: the **using_** class-template, which acts as a *reusable* base-class for smart references, and a tiny reflection facility, which is used for supporting user-defined types. Together, they provide a new building block with which powerful zero-overhead abstractions can be created, today.

1. Introduction

Smart references are a fundamental missing building block of the C++ language, and are a long-demanded feature by the C++ community [1]. They allow for creating objects that act as if they are other objects, like proxies, rebindable references, properties, strong typedefs, etc.

There currently exist several proposals for adding language support for smart references, (p0416r1, p0060r0, p0352r1). Despite the demand for language support, it does seem there is currently no consensus yet which direction to pursue.

The **smartref** library provides this missing building block, but instead as a pure-library solution. Furthermore, its syntax is based on the smart reference proposal from (p0352r1 - "Smart References through Delegation", which (arguably) leads to a very intuitive way of thinking about how to design smart references.

Overview

The core of the **smartref** library is the **using_** class-template, from which smart reference-classes can derive.

The using_ class-template:

- Parameterizes over the *delegate* type.
- Obtains the *delegate object* from the derived class.
- Defines members, corresponding to those found in that type.
- Member-*functions* forward the call to the delegate object.
- Member-*types* are aliases for the types defined in the delegate type.

The *smart reference* class:

- Defines a conversion function which returns the delegate object.
- Derives from the **using_** class, and thereby inherits the interface from the delegate type.

Example: a proxy class for on-demand loading from disk

```
// smartref library
template<typename T>
class proxy : public using_<T>
{
    T data_;

public:
    operator T &()
    {
        // ...lazy-load data_
        return data_;
    }
};

proxy<vector<double>> v = read_from_file(...);

v.push_back(3.141592654);

for (auto &x : v)
{
    x *= 2;
}
```

```
// p0352r1 - language-based
template<typename T>
class proxy : public using T
{
    T data_;

public:
    operator T &()
    {
        // ...lazy-load data_
        return data_;
    }
};
```

- `proxy` provides exactly the same interface as the wrapped `std::vector<double>`.
- Its definition differs only in three characters w.r.t. the language-based solution

Batteries included

Out of the box, the **using_** class-template defines member-functions and member-types corresponding to all those found in the data types defined by the STL. More importantly, this support is generic: any type satisfying (part of) the interface of an STL type, is also supported out of the box.

User-defined types

In order to support user-defined types, their members need to be explicitly registered.

For this, the **smartref** library comes with a tiny reflection facility, which provides a non-intrusive **REFLECTABLE** macro. By annotating the name of a member using this macro, this member will be picked up automatically by the **using_** class-template:

```
struct Foo
{
    int bar(bool a, unique_ptr<double> b) { ... }
};

REFLECTABLE(bar);

proxy<Foo> foo = ...;
auto x = foo.bar(true, make_unique<double>(3.141592654));
```

2. Technical description

The following sections will present a step-by-step description of how the **smartref** library was designed, and why some design decisions were chosen. It uses the **proxy** class from the introduction as an example, to finally arrive at the current design of the **using_** class.

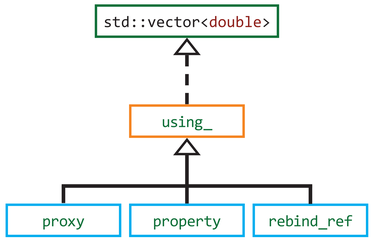
Defining a reusable base-class

```
class using_
{
    virtual operator std::vector<double> &() = 0;

    auto delegate()
    {
        return static_cast<std::vector<double> &>(*this);
    }

public:
    auto begin()    { return delegate().begin(); }
    auto end()      { return delegate().end();   }
};
```

```
class proxy : public using_
{
    std::vector<double> data_;

public:
    operator std::vector<double> &()
    {
        // ...lazy-load data_
        return data_;
    }
};
```

The proxy class:

- Inherits the interface of `std::vector<double>` from the **using_** class.
- **proxy** objects implicitly convert to `std::vector<double>`.
- Range-based for-loops are supported through `begin()` and `end()`.



Supporting arbitrary STL-style containers

There is no reason why we should only support `std::vector<double>`, as we can easily generalize to an arbitrary STL container type, or even our own container type that conforms to the STL api, e.g. `std::list<int>`:

```
template<typename Delegate>
class using_
{
    ...

    template<typename T>
    auto push_back(T &&value) -> decltype(delegate().push_back(std::forward<T>(value)))
    {
        return delegate().push_back(std::forward<T>(value));
    }
};
```

The zero-overhead principle

C++'s motto is "**Don't pay for what you don't use**". This also applies to the **smartref** library, and in two specific ways:

1. Member-function calls should have zero runtime overhead
2. The smart reference should add no memory overhead

The design above violates the first of these, whereas the second one is still maintained. The reason is the virtual conversion operator. Fortunately, this can be solved easily, by providing an opt-in mechanism:

```
template<typename Delegate, class Derived>
struct using_base
{
    operator Delegate &()
    {
        auto &derived = static_cast<Derived &>(*this); // <- downcast
        return static_cast<Delegate &>(derived);       // <- conversion
    }
};

template<typename Delegate>
struct using_base<Delegate, void>
{
    virtual operator Delegate &() = 0;
};

template<typename Delegate, class CRTP = void>
class using_ : public using_base<Delegate, CRTP>
{
    ...
};
```

Now, in order to get a zero-overhead smart reference class, the class can simply pass itself as the second template parameter:

```
template<typename T>
class proxy : public using_<T, proxy<T>>
{
    ...
};
```

Extensibility for user-defined types

In order to *generically* extend the **smartref** library to support user-defined types, we can have the **using_** class inherit from multiple **GenericMember** classes, each corresponding to an individual member-function.

```
template<typename Delegate, size_t slot>
struct GenericMember
{
};

template<typename Delegate, class CRTP = void>
class using_ : public using_base<Delegate, CRTP>
             , public GenericMember<Delegate, 0>
             , public GenericMember<Delegate, 1>
             , public GenericMember<Delegate, 2>
             , public GenericMember<Delegate, 3>
               // up to some high enough number
{
    ...
};
```

Now, we can declare each member-function separately:

```
template<typename Delegate>
struct GenericMember<Delegate, 0>
{
    auto bar() -> decltype(delegate(*this).bar())
    {
        return delegate(*this).bar();
    }
};
```

- Automatic discovery of supported member-functions.
- Works generically with any type supplying these member-functions.

Reducing the boiler-plate with a concise macro

```
#define REFLECTABLE(member)         // compile-time counter trick,     \
    template<typename Delegate>     // from CopperSpice (CppCon2015)   \
    struct GenericMember<Delegate, COUNTER()>                          \
    {                                                                  \
        auto member() -> decltype(delegate(*this).member())            \
        {                                                              \
            return delegate(*this).member();                           \
        }                                                              \
    };                                                                 \
                                                                       \
    INCREMENT_COUNTER();                                               \
```

3. Conclusions and future work

Advantages

- Uses existing familiar lookup rules.
- Can be used with existing compilers.
- Unlocks creativity to build new powerful abstractions today.
- Doesn't add more complexity to the C++ language.

Disadvantages

- Requires explicit marking any user-defined type's member reflectable.
- Compile-time performance is *possibly* worse than a language-based feature supported by the compiler.

Future work

- Member-fields.
- Compile-time performance.

Acknowledgements

I would like to thank the following people for their valuable feedback on the initial draft version of this poster: Eelco van den Berg, Markus Bürg, Andrei Chesaru, Eelco Cramer, Jan de Kleijn, Anita Laera, Matthijs Sypkens Smit, and Christa Valkering.

References

- [1] James Adcock: Request for Consideration - Overloadable Unary operator
- [2] Stroustrup, Dos Reis - Operator Dot (R3)
- [3] Gaunard, Kühl - Function Object-Based Overloading of Operator Dot
- [4] Tong, Vali - Smart References through Delegation (2nd revision)
- [5] Geller, Sermersheim - Compile-Time Counter Using Template & Constexpr Magic

URL: https://github.com/erikvalkering/smartref