PLAXIS

*essential for geotechnical professionals*

# Smart References

*There and Back Again*

https://github.com/erikvalkering/smartref

https://medium.com/@eejv

Erik Valkering

CppCon2018, September 25, 2018

# Agenda

- Motivation
- Library usage
- Library design
- Conclusion

Questions → after presentation

# Motivation

- Strong typedefs
- Proxies

# Motivation - Strong typedefs (1)

```
 1 using filename_t = string;
 2 using url_t      = string;
 3
 4 auto read(filename_t filename) { /* read from disk */     }
 5 auto read(url_t url)           { /* read from internet */ }
 6
 7 auto test()
 8 {
 9   auto filename = filename_t{"foobar.txt"};
10   auto url      = url_t{"http://foobar.com/"};
11
12   cout << "From disk [" << filename << "]: " << read(filename) << endl;
13   cout << "From web  [" << url      << "]: " << read(url)      << endl;
14 }
```

# Motivation - Strong typedefs (2)

```
1 template<typename T, typename tag>
2 class strong_typedef
3 {
4   /* magic */
5 };
```

# Motivation - Strong typedefs (3)

```
1 using filename_t = strong_typedef<string, class filename_t_tag>;
2 using url_t      = strong_typedef<string, class url_t_tag>;
3
4 auto read(filename_t filename) { /* read from disk */     }
5 auto read(url_t url)           { /* read from internet */ }
6
7 auto test()
8 {
9   auto filename = filename_t{"foobar.txt"};
10  auto url      = url_t{"http://foobar.com/"};
11
12  cout << "From disk [" << filename << "]: " << read(filename) << endl;
13  cout << "From web  [" << url      << "]: " << read(url)      << endl;
14 }
```

# Motivation - Proxies (1)

```
1 // Imagine one million objects, each 1 GB in size, or very tiny
2 using LargeModel = map<string, Proxy<vector<double>>>;
3 using SmallModel = map<string, array<double, 100>>;
4
5 template<typename Model>
6 auto averageData(Model &model, string key)
7 {
8   auto &data = model[key]; // <- Proxy<vector<double>>
9                            // or array<double, 100>
10  auto average = 0.0;
11  for (auto element : data)
12    average += element / data.size();
13
14  return average;
15 }
```

# Motivation - Proxies (2)

```
1  template<typename T>
2  class Proxy
3  {
4    /* magic */
5  };
6
7  Proxy<vector<double>>
8  Proxy<vector<int>>
9  Proxy<map<string, vector<float>>>
10 Proxy<SomeUserDefinedType>
```

# What is a smart reference?

# Smart Pointers vs Smart References

```
1 smart_ptr<Foo> foo = ...; // Mimicks "Foo *ptr = ..."
2 foo->bar();
3
4 auto *operator->()
5 {
6   ...
7 }
```

```
1 smart_ref<Foo> foo = ...; // Mimicks "Foo &ref = ..."
2 foo.bar();
3
4 auto &operator.() // <- DOES NOT EXIST
5 {
6   ...
7 }
```

# Interlude

# unified call syntax

## Given

```
auto v = vector{1, 2, 2, 1, 3, 2, 4, 5};
```

## Which do you find more readable?

```
auto s = sum(transform(filtered(unique(sorted(v)), is_even), squared));
```

## Or

```
auto s = v.sorted().unique().filtered(is_even).transform(squared).sum();
```

# Library usage

# Library usage - implicit conversion

```
1  template<typename T>
2  class Proxy
3  {
4    T data;
5
6  public:
7    operator T &() { /* lazy-load data... */ return data; }
8  };
9
10 auto foo(vector<double> &data) { ... }
11
12 auto proxy = Proxy<vector<double>>{...};
13 foo(proxy);
14
15
16
```

# Library usage - members (1)

```
1  template<typename T>
2  class Proxy : public using_<T>
3  {
4    T data;
5
6  public:
7    operator T &() { /* lazy-load data... */ return data; }
8  };
9
10 auto foo(vector<double> &data) { ... }
11
12 auto proxy = Proxy<vector<double>>{...};
13 foo(proxy);
14 proxy.begin();
15 proxy[0] = 0;
16 back_inserter(proxy);
```

# Library usage - members (2)

```
1  template<typename T>
2  class Prox
3  {
4    T data;
5
6  public:
7    operator
8  };
9
10 auto foo(v
11
12 auto proxy
13 foo(proxy)
14 proxy.begi
15 proxy[0] =
16 back_inserter(proxy);
```

```
1 /* automatically inserted */
2 using value_type    = ...;
3 using iterator_type = ...;
4 ...
5 auto begin() { ... }
6 auto end()   { ... }
7 auto size()  { ... }
8 ...
```

# Library usage - user-defined types

```
1  struct Person
2  {
3    auto first_name() {...}
4    auto last_name()  {...}
5  };
6
7  REFLECTABLE(first_name);
8  REFLECTABLE(last_name);
9
10 auto proxy_person = Proxy<Person>{...};
11
12 cout << "First name: " << proxy_person.first_name() << endl;
13 cout << "Last name:  " << proxy_person.last_name()  << endl;
```

# Library design

# Library design - a reusable base-class (1)

```
1  class using_
2  {
3    auto &delegate()
4    {
5      return static_cast<vector<double> &>(*this);
6    }
7
8  public:
9    virtual operator vector<double> &() = 0;
10
11   auto begin() { return delegate().begin(); }
12   auto end()   { return delegate().end();   }
13 };
```

# Library design - a reusable base-class (2)

```
1  class Proxy : public using_
2  {
3    vector<double> data;
4
5  public:
6    operator vector<double> &( )
7    {
8      // ...lazy-load data
9      return data;
10   };
11 };
```

# Library design - arbitrary STL containers

```
1  template<typename Delegate>
2  class using_
3  {
4    auto &delegate() { return static_cast<Delegate &>(*this); }
5
6  public:
7    virtual operator Delegate &() = 0;
8
9    auto begin() -> decltype(delegate().begin())
10   {
11     return delegate().begin();
12   }
13 };
```

# Library design - zero-overhead principle (1)

```
1 template<typename T>
2 class Proxy : public using_<T, Proxy<T>>
3 {
4   ...
5 };
```

# Library design - zero-overhead principle (2)

```
1  template<typename Delegate, class CRTP>
2  class using_
3  {
4    auto &delegate()
5    {
6      auto &derived = static_cast<CRTP &>(*this); // <- downcast
7      return static_cast<Delegate &>(derived);    // <- conversion
8    }
9
10   ...
11 };
```

# Library design - user-defined types (1)

```
1  template<typename Delegate, typename CRTP, size_t index>
2  struct Member {};
3
4  template<typename Delegate, typename CRTP>
5  struct Member<Delegate, CRTP, 0>
6  {
7    auto &delegate() { ... }
8
9    auto begin() -> decltype(delegate().begin())
10   {
11     return delegate().begin();
12   }
13 };
```

# Library design - user-defined types (2)

```
1 template<typename Delegate, class CRTP>
2 class using_ : public Member<Delegate, CRTP, 0>
3               , public Member<Delegate, CRTP, 1>
4               , public Member<Delegate, CRTP, 2>
5               , public Member<Delegate, CRTP, 3>
6               // up to some high enough number
7 {
8   ...
9 };
```

# Library design - user-defined types (3)

```
#define REFLECTABLE(member)                                    \
  template<typename Delegate, typename CRTP>                   \
  struct Member<Delegate, CRTP, __COUNTER__>                   \
  {                                                            \
    auto &delegate() { ... }                                   \
                                                               \
    auto member() -> decltype(delegate().member())             \
    {                                                          \
      return delegate().member();                              \
    }                                                          \
  };                                                           \
```

# Uncoupling reflection from the delegate object

# Library design - uncoupling reflection (1)

```
1  template<typename CRTP>
2  class Member<CRTP, 0>
3  {
4    auto &derived() { return static_cast<CRTP &>(*this); }
5
6  public:
7    auto begin()
8    {
9      auto invoker = [](auto &obj) { return obj.begin(); };
10     return derived().on_call(invoker);
11   }
12 };
```

# Library design - uncoupling reflection (2)

```
1  template<typename Delegate, typename CRTP>
2  class using_ : public ...
3  {
4    ...
5
6    template<typename Invoker>
7    auto on_call(Invoker invoker)
8    {
9      return invoker(delegate(*this));
10   }
11 };
```

# unified call syntax

# Library design - unified call syntax (1)

```
1  template<typename CRTP>
2  class Member<CRTP, 0>
3  {
4    ...
5    auto begin()
6    {
7      auto invoker_member    = [](auto &obj) { return obj.begin(); };
8      auto invoker_nonmember = [](auto &obj) { return begin(obj);  };
9
10     struct ComposedInvoker : decltype(invoker_member)
11     {
12       auto as_nonmember() { return decltype(invoker_nonmember){}; }
13     };
14
15     return derived().on_call(ComposedInvoker{});
16   }
17 };
```

# Library design - unified call syntax (2)

```
1  template<typename T>
2  class UnifiedCallSyntax : public using_<T, UnifiedCallSyntax<T>>
3  {
4    ...
5
6    template<typename Invoker>
7    auto on_call(Invoker invoker)
8    {
9      auto invoker_nonmember = invoker.as_nonmember();
10     auto result            = invoker_nonmember(delegate(*this));
11
12     return UnifiedCallSyntax{result};
13   }
14 };
```

# Library design - unified call syntax (3)

```
1  auto sort(const auto &container) { ... }
2  auto unique(const auto &container) { ... }
3  ...
4
5  template<typename T>
6  using $ = UnifiedCallSyntax<T>;
7
8  auto v = vector{1, 2, 2, 1, 3, 2, 4, 5};
9
10 auto s = $(v).sort()
11              .unique()
12              .filter(is_even)
13              .transform(squared)
14              .sum();
```

# Conclusion

- We presented a design for creating smart references
  - Using today's C++
  - Generically support arbitrary classes
  - Support for user-defined classes
  - Zero-overhead memory & runtime
  - Advanced use cases through uncoupled design

Implementation available at:
https://github.com/erikvalkering/smartref

PLAXIS

*essential for geotechnical professionals*

# Thank you

www.plaxis.nl

Plaxis bv
Tel: +31 (0)15 251 7720
Fax: +31 (0)15 257 3107

P.O. Box 572
2600 AN Delft
The Netherlands

Plaxis AsiaPac Pte Ltd
Tel: +65 6325 4191
Fax: +65 6271 6066

16 Jalan Kilang Timor
#05-08 Redhill Forum
159308 Singapore

# This presentation has been co-sponsored by

MINDESK®


The first VR interface in the world 100% integrated within your CAD software.

https://www.mindeskvr.com/