Make and Forward Consultables and Delegates

Sometimes forwarding to a contained object requires lots of boilerplate code. Nicolas Bouillot introduces consultables and delegates to automate this.

Separation of functionalities, code reuse, code maintainability: all these concerns seem critical. When writing a C++ class, *selective* delegation and inheritance are tools enabling both separation of functionality implementation and code reuse. Before introducing the making and forwarding of Consultable and Delegate, let us review some points that make selective delegation and inheritance two very different tools in the hand of the programmer. In particular, it will be explained why they might be seen as poorly scaling tools with increase of code base, and why the new approach *Consultables* and *Delegates* presented here scales better.

With public inheritance, public methods from the base class are made available directly through the invocation of the inherited methods. While this makes available an additional functionality without extra code, inheriting from multiple functionalities need cleverly written base classes in order to avoid members and methods collisions. As a result the more base classes are inherited from, the more collisions are probable, including the probability of inheriting multiple times from the same base class (the diamond problem) and related issues.

With traditional C++ delegation, called here *selective delegation*, the added functionality is a private class member (the delegate), which can be accessed by the user thanks to wrappers in allowing to access the member's methods. This scales well with the number of added functionalities. One can even have two functionalities of the same type used as delegate, which cannot be achieved with inheritance. However, manually writing wrappers does not scale well with the increased use of delegation: once the signature of a method is changed in the delegate, the wrappers need to be manually updated.

Public inheritance and selective delegation allows access to delegates, but what about read/write access? With inheritance, all public methods, regardless of their constness are made available to the user. There is no option to give access to `const` only methods, and accordingly reserve the write access to the member for internal use while giving read access to the user. With selective delegation, the access rights to the functionality are given through wrapper, giving full control to the programmer who can do read/write access manual if desired. With the *Consultables* and *Delegates* presented here, templated wrappers are automatically generated according to the `const`ness, enabling or not the methods with the desired `const`ness.

Let us take an example. A class (delegator) is having a `std::vector` (delegate) for internal use (write access). However, it is wanted to give a read access to the user (delegator owner). In this case, the delegator inheriting from `std::vector` is not an option, and selective delegation requires the programmer wrap all `const` methods and operators for the user, or a subset (more probable) of methods. When wrapping manually, is it considered necessary to include access to `cbegin` and `cend`? Probably not if the user does not use them when writing the wrapper. With templated wrapper generation, no manual selection is required and wrappers are generated only if invoked by a user. As a result, no wrappers with errors, no wrapper forgetting and no need to rewrite wrappers if the contained type changes (moving from a `string` vector to an `int` vector).

Let us go further: what about forwarding delegation? The class (delegator) owning the delegate is used by an other class (delegator owner). With selective delegation, the wrapper needs to be rewritten in order to make initial delegate available to the user of the delegator owner. Once more, manual wrapper should probably be avoided.

The following is presenting the use and implementation[1] of *Consultable* and *Delegate*. It is built with C++11 template programming. As briefly said earlier, *Consultable* makes available all `public const` methods for safe access, and *Delegate* that is making all public delegate methods available. The benefits of this approach is 1) no specific wrappers, so less code to write and maintain, 2) a change in the delegate code becomes immediately available at the end caller, producing a more easily maintainable code base and 3) non-const method access of a delegate can be blocked in any forwarding class by forwarding the delegate as a Consultable.

Selective delegation vs. Consultables & Delegates

Brief history: the Gang of Four [GoF95] describes delegation as consisting of having an requested object that delegates operations to a delegate object it is owning. This method is comparable to subclassing where inheritance allows for a subclass to automatically re-uses code (no wrapper) in a base class. While delegation requires the implementation of wrappers, subclassing has been seen as providing poor scaling and tends to lead to code that is hard to read, and therefore hard to correct and maintain [Reenskaug07]. As a rule of thumb, Scott Meyers proposes to use inheritance only when objects have an 'is-a' relationship [Meyers05]. It is, however, tempting for programmers to choose inheritance instead of delegation since writing wrappers is a tedious task and can still be error-prone.

Delegation through automatic generation of wrappers for *selected* methods has been extensively proposed and implemented[2]. While this gives a full control of which method are available, this also requires specification of each delegated methods along the possible classes that delegate,

Nicolas Bouillot Nicolas is a research associate at the Society for Arts and Technology (SAT, Montreal, Canada). He likes C++ programming, team-based working, writing research papers, networks and distributed systems, data streaming, distributed music performances, teaching, audio signal processing and many other unrelated things.

1. Source code and examples are available at https://github.com/nicobou/cpp_make_consultable
2. (Accessed March 2015)
http://www.codeproject.com/Articles/11015/The-Impossibly-Fast-C-Delegates
http://www.codeproject.com/Articles/7150/Member-Function-Pointers-and-the-Fastest-Possible
http://www.codeproject.com/Articles/18886/A-new-way-to-implement-Delegate-in-C
http://www.codeproject.com/Articles/384572/Implementation-of-Delegates-in-Cplusplus11
http://www.codeproject.com/Articles/412968/ReflectionHelper

The goal of the approach proposed here is to avoid specifying the delegate methods

subdelegate, sub-subdelegate, etc. This may result in laborious specification in multiple files, leading to duplication in the code base. In turn, safety remains in the hands of the developer who decides a non-`const` method can be exposed.

The goal of the approach proposed here is to avoid specifying the delegate methods, but instead makes available all or `const` only methods. This is achieved by making a class member `Consultable` or `Delegate` that is accessed through a programmer specified delegator method. The use of a Consultable – invoking a method of the delegate – by the requester is achieved by invoking the consult method, whose arguments are the delegate method followed by the arguments.

Consultable

Usage

Listing 1 shows how consultables are used. Two `Widget`s will be owned by a `WidgetOwner`. These two members are made consultable in `WidgetOwner` (lines 14 and 15) and will be accessed respectively with the `consult_first` and `consult_second` methods. `Make_consultable`, as explained in 'Implementation' below, is a macro that internally declares templated consult methods. Its arguments are the type of the delegate, a reference to the delegate member and the `consult` method.

In the main function, the WidgetOwner object `wo` is instantiated and the Widget const method `get_name` is invoked on both delegates (lines 24 and 25). Note the comment in line 28 showing an example of a use of `consult_first` with the non const `set_name` method, which does not compile.

Forwarding Consultable

As seen before, an object is made consultable and accessible through a consult method. Accordingly, a class composed of delegate owner(s) can access the delegate methods. However, this class does not have access to the reference of the original delegate, making impossible to apply `Make_consultable`. In this case, `Forward_consultable` allows selection of the consult method and *forwards* it to the user. A forwarded consultable can be re-forwarded, etc.

Listing 2 shows how it is used: `WidgetOwner` is making `first_` and `second_` consultables. Box is owning a `WidgetOwner` and forward access to the consultables using `Forward_consultable` (line 20 and 21). Then, a Box Owner can access the consultable through `fwd_first`, a `Box` method installed by the forward (line 28).

Overloads

The use of consultable requires the user to pass a pointer to the delegate method as argument. Accordingly, overloaded delegate members need more specification for use with consult methods. As shown in Listing 3, this is achieved giving return and argument(s) types as template parameters (line 14 and 15). Overload selection can also be achieved using static casting the member pointer (line 21). This is however more verbose and might not be appropriate for use.

Enabling setters

Although a *Consultable* makes available all `public const` methods, it may need to have a setter that can be safely used. For instance, *consultable*

Consultable declaration and use. This illustrates how `get_name` is accessed by a `WidgetOwner` without explicit declaration of it in the `WidgetOwner` class. At line 14, the macro `Make_consultable` is actually declaring several methods named `consult_first` that will make consultation available to the user.

```
 1 class Widget {
 2  public:
 3   Widget(const string &name): name_(name){}
 4   string get_name() const { return name_; }
 5   string hello(const string str) const {
       return "hello " + str;};
 6   void set_name(const string &name) {
       name_ = name; }
 7
 8  private:
 9    string name_{};
10 };
11
12 class WidgetOwner {
13  public:
14   Make_consultable(Widget, &first_,
       consult_first);
15   Make_consultable(Widget, &second_,
       consult_second);
16
17  private:
18   Widget first_{"first"};
19   Widget second_{"second"};
20 };
21
22 int main() {
23   WidgetOwner wo; // print:
24   cout << wo.consult_first(&Widget::get_name)
     // first
25        << wo.consult_second(&Widget::get_name)
          // second
26        << wo.consult_second(&Widget::hello,
          "you")  // hello you
27        << endl;
28   // compile time error (Widget::set_name
     is not const):
29   // wo.consult_first(&Widget::set_name,
     "third");
30 }
```

Listing 1

Forwarding consultable example.

```
 1 class Widget {
 2  public:
 3   Widget(const string &str): name_(str){}
 4   string get_name() const {return name_;}
 5  private:
 6   string name_;
 7 };
 8
 9 class WidgetOwner {
10  public:
11   Make_consultable(Widget, &first_,
     consult_first);
12   Make_consultable(Widget, &second_,
     consult_second);
13  private:
14   Widget first_{"First"};
15   Widget second_{"Second"};
16 };
17
18 class Box {
19  public:
20   Forward_consultable(WidgetOwner, &wo_,
     consult_first, fwd_first);
21   Forward_consultable(WidgetOwner, &wo_,
     consult_second, fwd_second);
22  private:
23   WidgetOwner wo_;
24 };
25
26 int main() {
27   Box b{};
28   cout << b.fwd_first(&Widget::get_name)
     // prints First
29        << b.fwd_second(&Widget::get_name)
        // prints Second
30        << endl;
31 }
```

Listing 2

could provide a setter that configure the refresh rate of an internal measure in a thread, or a setter for registering a callback function. Since **`Make_consultable`** does not feature selective inclusion of a non-const methods, a possible way for setter inclusion is to declare it const concerned member **`mutable`**. This is illustrated with Listing 4, where the enabling of a setter is obtained from inside the delegate class.

This approach seems to be more appropriate than introducing selective delegation from the delegator. First, the selection of a method from the delegator would break safety: it will be potentially forwarded and then accessed by multiple class, allowing internal state modification from several classes. In this case, the setter may need internal mechanism for being safe to be accessed (mutex, ...), which should probably be ensured from the delegate itself. Second, the use of **`mutable`** and **`const`** for the member setter is making explicit the intention of giving a safe access and is accordingly indicating to the user the method is available for consultation directly from the delegate header.

Implementation

An extract of implementation of both **`Make_consultable`** and **`Forward_consultable`** are presented here. Although not presented here, the available source code hosted on github provides additional overloads for methods returning void.

Listing 5 presents the **`Make_consultable`** implementation. The use of a macro allows for declaring the consultation methods. Accordingly, a consult method (**`_consult_method`**), which name is given by a macro argument, can be specified as public or protected. The declaration consists of two overloads that are distinguished by their constness. More

Overloaded methods in the delegate class. Types from overloaded method are given as template parameters in order to select the wanted implementation.

```
 1 class Widget {
 2  public:
 3  ...
 4   string hello() const { return "hello"; }
 5   string hello(const std::string &str) const {
       return "hello " + str; }
 6 };
 7
 8 ...
 9
10 int main() {
11   WidgetOwner wo{};
12   // In case of overloads, signature types give
        as template parameter
13   // allows to distinguishing which overload to
        select
14   cout << wo.consult_first<string>
        (&Widget::hello)  // hello
15        << wo.consult_first<string,
          const string &>(
16             &Widget::hello, std::string("ho"))
               // hello ho
17        << endl;
18
19        // static_cast allows for more verbosely
             selecting the wanted
20        cout << wo.consult_first(
21             static_cast<string(Widget::*)
22                         (const string &)
                            const>(&Widget::hello),
23                          "you")  // hello you
24             << endl;
25 }
```

Listing 3

Enabling setters in the delegate class. This is achieved by making the setter a const method (line 5), and accordingly making the member mutable (line 9). The setter is used in order to pass a lambda (lines 23–25) function that will be stored by the Widget (line 6). As a result, the setter is explicitly enabled when Widget is made consultable. This is also made explicit to the programmer who is reading the Widget header file.

```
 1 class Widget {
 2  public:
 3   // make a setter consultable from inside
     the delegate
 4   // set_callback needs to be const and cb_
     needs to be mutable
 5   void set_callback(std::function<void ()> cb)
     const {
 6     cb_ = cb;
 7   }
 8  private:
 9   mutable std::function<void()> cb_{nullptr};
10 };
11
12 class WidgetOwner {
13  public:
14   Make_consultable(Widget, &first_,
     consult_first);
15
16   private:
17   Widget first_{};
18 };
```

Listing 4

```
19
20 int main() {
21   WidgetOwner wo{};
22   // accessing set_name (made const from the
     Widget)
23   wo.consult_first(&Widget::set_callback,
     [](){
24       std::cout << "callback" << std::endl;
25     });
26 }
```

Listing 4 (cont'd)

particularly, the non-const overload fails to compile with a systematic **static_assert** when this overload is selected by type deduction, disabling accordingly the use of delegate non-const methods. The consult method (line 13) uses a variadic template for wrapping any const methods from the delegate (**fun**), taking the method pointer and the arguments to pass at invocation. Notice the types of fun arguments (**ATs**) and the types of the consult method arguments (**BTs**) are specified independently, allowing a user to pass arguments which does not have the exact same type, allowing them to pass a **char *** for an argument specified as a **const string &**.

The delegate type is saved for later reuse when forwarding (lines 6 and 7). This is using macro concatenation **##** in order generate a type name a forwarder can find, i.e. the consult method name concatenated with the string **Consult_t**.

Implementation of **Make_consultable**. Overload resolution for **_consult_method** select delegate member according to their **const**ness. If not **const**, compilation is aborted with **static_assert**.

```
 1 #define Make_consultable(_member_type,          \
 2                          _member_rawptr,        \
 3                          _consult_method)       \
 4                                                 \
 5   /*saving consultable type for the             \
     forwarder(s)*/                                \
 6   using _consult_method##Consult_t = typename   \
 7       std::remove_pointer<std::decay
           <_member_type>::type>::type;            \
 8                                                 \
 9   /* exposing T const methods accessible by T
        instance owner*/                           \
10   template<typename R,                          \
11            typename ...ATs,                     \
12            typename ...BTs>                     \
13   inline R _consult_method(R
     (_member_type::*fun)(ATs...) const,           \
14                           BTs ...args) const {\
15     return ((_member_rawptr)->*fun)
       (std::forward<BTs>(args)...);               \
16   }                                             \
17                                                 \
18   /* disable invocation of non const*/          \
19   template<typename R,                          \
20            typename ...ATs,                     \
21            typename ...BTs>                     \
22   R _consult_method(R
       (_member_type::*function)(ATs...),          \
23                     BTs ...) const {            \
24     static_assert(std::is_const<decltype
       (function)>::value,                         \
25       "consultation is available for const
        methods only");                            \
26     return R();  /* for syntax only since
       assert should always fail */                \
27   }                                             \
```

Listing 5

This brings us to the implementation of **Forward_consultable** (Listing 6). As with **Make_consultable**, it is actually an inlined wrapper generator using variadic template. However, it has no reference to the delegate, but only to the consult method it is invoking at line 18. The delegate type in obtained from the previously saved member type (line 7 to 9) and used for invoking the consult method (line 19). Again, the delegate type is saved for possible forwarders of this forward (line 7).

Make & Forward Delegate

Usage

The Delegate presented here is similar to Consultable, except that non-const methods are also enabled. They are also forwardable as Consultable, blocking the access to non-const methods of the owner of the forwarded. Listing 7 illustrates how access to non-const method can be managed and blocked when desired: **hello** is a non-const method in Widget. **widgetOwner** is Making two Delegates **first_** and **second_** (lines 13 & 14). hello is accordingly available from the owner of a **WidgetOwner** (lines 32 & 33). However, **Box** is forwarding access to **first_** as consultable (line 23) and access to **second_** as delegate (line 24). As a result, the owner of a **Box** have access to non-const method of **second_** (line 39), but access to const only methods of **first_** (line 37).

Implementation

The core implementation of **Make_Delegate** is actually the same as Consultable. In the source code, consultable and delegate is chosen with a flag that enables or disables access to non-const methods[3]. **Make_consultable** and **Make_delegate** are actually macros that expand to the same macro with the appropriate flag.

Listing 8 is presenting how this flag is implemented in the **Make_access** macro. This template is selected when a non-const pointer to the delegate is given (**fun** is not a const member). The flag is a non-type template parameter **flag** (line 15) which value is determined by the concatenation of macro parameters **_consult_method** and **_access_flag**. This value is tested (**static_assert** line 18) against the pre-defined enum values (line 6-9), enabling access to non-const methods or not at compile

Implementation of **Forward_consultable**.

```
 1 #define Forward_consultable(_member_type,     \
 2                             _member_rawptr,   \
 3                             _consult_method,  \
 4                             _fw_method)       \
 5                                               \
 6   /*forwarding consultable type for other
       forwarder(s)*/                            \
 7   using _fw_method##Consult_t = typename      \
 8       std::decay<_member_type>::type::        \
 9       _consult_method##Consult_t;             \
10                                               \
11   template<typename R,                        \
12            typename ...ATs,                   \
13            typename ...BTs>                   \
14   inline R _fw_method(                        \
15       R( _fw_method##Consult_t
             ::*function)(ATs...) const,         \
16       BTs ...args) const {                    \
17     return (_member_rawptr)->                 \
18         _consult_method<R, ATs...>(           \
19             std::forward<R(
               _fw_method##Consult_t ::*)
               (ATs...) const>(                  \
20                 function),                    \
21             std::forward<BTs>(args)...);      \
22   }                                           \
```

Listing 6

3. For expected improvement of clarity, Listing 5 is a simplification of actual code, hiding the flag mechanism.

time. Accordingly, **_access_flag** must take one of the two following string: **non_const** or **const_only**.

The implementation of **Forward_delegate** and **Forward_consultable** is also using the same flag mechanism for disabling or not the use of non-const methods.

Summary and discussion

Consultable and *Delegate* has been presented, including how they can be used with overloaded methods inside the delegate class, how specific setters can be enabled with for consultable. Forwarding is also presented, allowing to make available delegate methods to the user through any number of classes with access rights (public const methods only or all public methods).

Their implementation is based on C++11 template programming and macros inserted inside the class declaration, specifying the member to delegate and the name of the access method to generate to the user. Examples and source code are available on github and have been compiled and tested successfully using with gcc 4.8.2-19ubuntu1, clang 3.4-1ubuntu3 & Apple LLVM version 6.0.

Consultable is currently used in production code where it was found very useful for refactoring delegate interface for adaptation or experimentation. This is also reducing size of code in classes that use delegation/consultation extensively, and makes clearer from the code what can be done. Without *Consultables*, the wrappers from the various delegates makes the code very noisy and hard to read. Additionally, this project makes use of **Forward_consultable_from_map**, which has not been presented here since very specific. A more generic way of Forwarding from containers should be designed and developed in order to let the user manage container searching, along with error handling. ■

Acknowledgement

This work has been done at the Société des Arts Technologiques and funded by the Ministère de l'Économie, de l'Innovation et des Exportations (Québec, Canada).

References

[GoF95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[Meyers05] Scott Meyers. *Effective C++: 55 Specific Ways to Improve Your Programs and Designs* (3rd Edition). Addison-Wesley Professional, 2005.

[Reenskaug07] Trygve Reenskaug. *Computer Software Engineering Research*, chapter 'The Case for Readable Code'. Nova Science Publishers, Inc., 2007.

Make and forward delegate example.

```
 1 class Widget {
 2  public:
 3   std::string hello(const std::string &str) {
 4     last_hello_ = str;
 5     return "hello " + str;
 6   }
 7  private:
 8   std::string last_hello_{};
 9 };
10
11 class WidgetOwner {
12  public:
13   Make_delegate(Widget, &first_, use_first);
14   Make_delegate(Widget, &second_, use_second);
15
16  private:
17   Widget first_{};
18   Widget second_{};
19 };
20
21 class Box {
22  public:
23   Forward_consultable(WidgetOwner, &wo_,
     use_first, fwd_first);
24   Forward_delegate(WidgetOwner, &wo_,
     use_second, fwd_second);
25  private:
26   WidgetOwner wo_;
27 };
28
29 int main() {
30   WidgetOwner wo{};
31   // both invocation are allowed since first_
     and second are delegated
32   cout << wo.use_first(&Widget::hello, "you")
     << endl;   // hello you
33   cout << wo.use_second(&Widget::hello, "you")
     << endl;   // hello you
34
35   Box b{};
36   // compile error, first_ is now a consultable:
37   // cout << b.fwd_first(&Widget::hello, "you")
     << endl;
38   //  OK, second_ is a delegate:
39   cout << b.fwd_second(&Widget::hello, "you")
     << endl;   // hello you
40 }
```

Listing 7

Extract of the core implementation of both **Make_Delegate** and **Make_consultable**. **_access_flag** is used in order to enable/disable the use of non const methods. This flag is set to **non_const** by **Make_delegate** and to **const_only** by **Make_delegate**.

```
 1 #define Make_access(_member_type,              \
 2                     _member_rawptr,            \
 3                     _consult_method,           \
 4                     _access_flag)              \
 5                                                \
 6   enum _consult_method##NonConst_t {           \
 7     _consult_method##non_const,                \
 8         _consult_method##const_only            \
 9         };                                     \
10                                                \
11   /* disable invocation of non-const
     if the flag is set*/                         \
12   template<typename R,                         \
13            typename ...ATs,                    \
14            typename ...BTs,                    \
15            int flag=
              _consult_method##_access_flag>      \
16   inline R _consult_method
     (R(_member_type::*fun)(ATs...),              \
17      BTs ...args) {                            \
18     static_assert(flag ==
       _consult_method##NonConst_t::              \
19     _consult_method##non_const,                \
20     "consultation is available for const
       methods only "                             \
21     "and delegation is disabled");             \
22     return ((_member_rawptr)->*fun)
       (std::forward<BTs>(args)...);              \
23   }                                            \
```

Listing 8