A Callback Class

Doug Young

This paper briefly describes an alternate approach for using C++ member functions with Xt callbacks. This material was originally written as an appendix to *Object-Oriented Programming with C++ and OSF/Motif* f, Prentice Hall, 1992, but was cut from the final manuscript in the interest of simplicity and consistency.

The approach used throughout *Object-Oriented Programming with C++ and OSF/Motif* for dealing with member functions and widget callbacks requires two functions. One function is declared as a static member function and registered as a Motif callback using `XtAddCallback()`. An object's `this` pointer is provided as client data when the callback is registered. When this static member function is called, it retrieves the instance pointer from its client data argument and calls the second function, which can be a normal member function, relative to that object.

This approach leads to many member functions that are very similar. For example, almost every callback function looks like:

```
void SomeClass::memberFunctionCallback ( Widget,
                                         XtPointer clientData,
                                         XtPointer )
{
    SomeClass *obj = (SomeClass *) clientData;
    obj->memberFunction();
}
```

The only difference between various static functions used as callbacks is the class to which the function belongs, and the exact member function being called. The form is identical in all cases

Because the form of all these functions is so similar, it is possible to hide some of the details of this approach. One way to do this is to capture the entire mechanism in a C++ class. This paper examines an implementation of such a class, the Callback class.

The Callback class has some similarities to the Cmd class described in Chapter 7 of *Object-Oriented Programming with C++ and OSF/Motif*. However, the implementation is a bit different. Basically, every class that wants to use a Callback object must create a new derived class that overrides an `execute()` member function declared by the Callback class. This member function is called by a static member function, registered by the Callback base class as the actual callback function. The `execute()` function is expected to call a normal member function.

This scheme requires only a single static member function per class, and only a single `execute()` member function. The technique allows normal member functions to be registered as pseudo-callbacks. However, the approach adds one more level of indirection than the alternate approach.

Let's start by looking at the declaration of the Callback class. The entire mechanism is implemented in a header file, using macros and inline functions. The file Callback.h is written as follows:

```
/////////////////////////////////////////////////////////
// Callback.h: A class that encapsulates callbacks
/////////////////////////////////////////////////////////
#ifndef CALLBACK_H
#define CALLBACK_H

#include <Xm/Xm.h>
#include <generic.h>

 class Callback {

   protected:

     Widget    _w;          // Widget for which callback is registered
     String    _name;       // The callback name,
                            // needed to remove callback
     XtPointer _clientData; // Xt's idea of client data is used
                            // for the this pointer,
                            // so replace it with our own.

     // Constructor registers callback and saves other info
     // that might be needed later as data members

     Callback ( Widget w, String name, XtPointer clientData )
     {
         XtAddCallback ( w,
                         name,
                         &Callback::callbackStub,
                         (XtPointer) this );
         _w          = w;
         _name       = name;
```

```
        _clientData = clientData;
    }

    // Destructor removes the callback, using save data members

    ~Callback()
    {
        XtRemoveCallback ( _w,
                           _name,
                           &Callback::callbackStub,
                           (XtPointer)this );
    }

    // Execute must be overriden by derived classes

    virtual void execute ( Widget, XtPointer, XtPointer ) = 0;

  private:

    // Generic callback function works for all callbacks

    static void callbackStub ( Widget    w,
                               XtPointer clientData,
                               XtPointer callData )
    {
        Callback *obj = (Callback *) clientData;
        obj->execute ( w, obj->_clientData, callData );
    }
};
```

The Callback class declares three protected data members, the widget with which the callback is registered, the name of the callback, and a pointer that can be used to point to some arbitrary data. Recall that the static member function scheme uses the client data supported by `XtAddCallback()` to pass the `this` pointer to the callback function. Although this scheme still needs to have the `this` pointer, wrapping the callback in a class hides the process and provides a place to keep some additional data on a per callback basis. This allows us to make the client data argument available to programmers again.

The Callback constructor expects a widget, the name of a callback and a pointer to any client data. It registers `callbackStub()`, which is a static member function defined by the Callback class, as the named callback function for the given widget. The Callback constructor also retains the widget, the name, and the client data as data members in the Callback object.

The destructor is very simple. It uses the callback name and widget kept in the object and calls `XtRemoveCallback()` to remove the callback registered in the constructor.

The function `callbackStub()` is registered for all callbacks, regardless of the type. It is declared as an inline static member function, and looks much like the other callback function used throughout this book. It retrieves the `this` pointer provided as client data when the callback was registered, casts it to the appropriate type, and calls the `execute()` member function for this instance. Because all callbacks use this same member function, the type of the object passed as client data will always be Callback. This makes this approach slightly more type safe than the alternate approach, in spite of the cast, which is still needed.

The `execute()` member function is declared as a pure virtual function, and must therefore be defined by derived classes. This function calls the member function registered as the pseudo-callback.

Now lets' take a first look at how this class is used. The Callback class cannot be used directly. A derived class must be created from every class that needs to register callbacks. It is easiest to just look at an example. The following ControlCallback class could be used with the Control class described in Chapter 2 of *Object-Oriented Programming with C++ and OSF/Motif*. We could declare the ControlCallback class as follows:

```
/////////////////////////////////////////////////////////////
// ControlCallback.h: Callback class to be used with Control
//////////////////////////////////////////////////////////////
#ifndef CONTROLCALLBACK_H
#define CONTROLCALLBACK_H

#include "Callback.h"

// Define ControllPMF as a type

typedef void ( Control::*ControlPMF  )( Widget, XtPointer, XtPointer );

// Declare the ControlCallback class as a derived calss of Callback

class  ControlCallback : public Callback {

  public:

   // Constructor takes a pointer to an instance of Control,
   // and a pointer to a Control member function, in addition
   // to the arguments requires by the Callback constructor

    ControlCallback  ( Control    *obj,
                       ControlPMF  pmf,
                       Widget      w,
                       String      callbackName,
                       XtPointer   clientData ) :
                              Callback ( w, callbackName, clientData )
    {
        _obj = obj; // Keep the instance pointer around
        _pmf = pmf; // Remember the member function pointer
    }

  protected:

    Control     *_obj;  // Stores a Control object
```

```
    ControlPMF   _pmf;  // A pointer to a Control member function

  private:

    // execute() is called from Callback class, indirectly
    // when callback is invoked. Use the stored Control instance
    // to call the member function. Pass the widget, calldata
    // and client data, as stored in this object.

    virtual void execute ( Widget w,
                           XtPointer clientData,
                           XtPointer callData )
    {
         ( _obj->*_pmf )( w, clientData, callData );
     }
};
```

To use this class, we need to instantiate a ControlCallback object with the appropriate arguments for each desired callback. Notice that only one ControlCallback class is needed regardless of how many widgets or how many different member functions are supported by the Control class.

However, notice that this ControlCallback class can only be used with the Control class. Using this technique with other classes would require a new derived class, with the appropriate members defined to point to instances of each new class.

This seems like a lot of work, just to avoid writing a simple static member function. Fortunately, we can use a simple technique to reduce the amount of work require down to a single line of code. Let's return to the file Callback.h and look at the following macro, which is defined following the Callback class declaration:

```
#define Callbackdeclare(CLASS)                                      \
class CLASS;                                                        \
                                                                    \
typedef void (CLASS::*name2(CLASS,PMF))( Widget,                    \
                                         XtPointer,                 \
                                         XtPointer );               \
                                                                    \
class name2(CLASS,Callback) : public Callback {                     \
                                                                    \
 public:                                                            \
                                                                    \
   name2(CLASS,Callback)( CLASS* obj,                               \
                          name2(CLASS,PMF) pmf,                     \
                          Widget w,                                 \
                          String callbackName,                      \
                          XtPointer clientData ) :                  \
                 Callback ( w,                                      \
                            callbackName,                           \
                            clientData )                            \
   {                                                                \
```

```
81      _obj = obj;                                                     \
82      _pmf = pmf;                                                     \
83   }                                                                  \
84                                                                      \
85    protected:                                                        \
86                                                                      \
87      CLASS* _obj;                                                    \
88      name2(CLASS,PMF) _pmf;                                          \
89                                                                      \
90    private:                                                          \
91                                                                      \
92      virtual void execute ( Widget w,                                \
93                             XtPointer clientData,                    \
94                             XtPointer callData )                     \
95  {                                                                   \
96          ( _obj->*_pmf )( w, clientData, callData );                 \
97      }                                                               \
98  };                                                                  \
99                                                                      \
100   name2(CLASS,Callback) *addCallback ( Widget w,                    \
101                                        String name,                 \
102                                        name2(CLASS,PMF) pmf,        \
103                                        XtPointer clientData )       \
104   {                                                                 \
105       name2(CLASS,Callback) *cb =                                   \
106                     new name2(CLASS,Callback)( this,                \
107                                                pmf,                 \
108                                                w,                   \
109                                                name,                \
110                                                clientData );        \
111      return cb;                                                     \
112   }
113  #endif
```

This macro captures the format of the ControlCallback class we just discussed. This macro uses the `name2()` macro found in the file generic.h to concatenate two symbols together to generate one new symbol. If we were to process the statement

```
CallbackDeclare(Control)
```

through the C++ preprocessor, the result would be approximately the same as the ControlCallback class implemented earlier. There is one additional feature of this macro, the `addCallback()` function defined starting on line 100. This is a convenience function that instantiates a Callback object and provides the this pointer to the constructor.

This macro is intended to be included in the declaration of a class. This makes the `addCallback()` function a member function of the declaring class.

With this macro defined, it is very easy to use the Callback class. Let's look at how it could be used with the Control class described in Chapter 2 of *Object-Oriented Programming with C++ and*

OSF/Motif. The class declaration is nearly identical to the declaration in that book, except that the class has no static member functions, and the `Callbackdeclare` macro is included inside the private portion of the class. Notice that the `stop()` and `start()` member functions support the arguments expected by the Callback class as well.

```
//////////////////////////////////////////////////////////////////////
// Control.h: A start/stop pair of buttons for the stopwatch program
//////////////////////////////////////////////////////////////////////
#ifndef CONTROL_H
#define CONTROL_H
#include "BasicComponent.h"
#include “Callback.h”

class Timer;
class Stopwatch;

class Control : public BasicComponent {

  private:

    void start ( Widget, XtPointer, XtPointer );
    void stop ( Widget, XtPointer, XtPointer );

    Callbackdeclare(Control)

 protected:

    Timer     *_timer;       // The timer controlled by this class
    Stopwatch *_stopwatch;   // The stopwatch containing this control
    Widget     _startWidget; // The start button
    Widget     _stopWidget;  // Stop button

  public:

    Control ( Widget, char * , Stopwatch *, Timer * );
};
#endif
```

The Control constructor is much the same as before. However, this version registers normal member functions as callbacks using the `addCallback()` member function generated by the `Callbackdeclare()` macro. Notice that the client data is specified as `NULL` in this case, and that no `this` pointer is required.

```
//////////////////////////////////////////////////////////////////////
// Control.C: A start/stop pair of buttons for the stopwatch program
//////////////////////////////////////////////////////////////////////
#include "Control.h"
#include <Xm/Xm.h>
```

```
#include <Xm/RowColumn.h>
#include <Xm/PushB.h>
#include "Timer.h"
#include "Stopwatch.h"

Control::Control ( Widget    parent,
                   char      *name,
                   Stopwatch *stopwatch,
                   Timer     *timer ) : BasicComponent ( name )
{

    _timer = timer;  // Keep a pointer to the timer.

    _stopwatch = stopwatch;

    // Create the component's widget tree

    _w = XmCreateRowColumn ( parent, _name, NULL, 0 );

    _startWidget = XtCreateManagedWidget ( "start",
                                           xmPushButtonWidgetClass,
                                           _w,  NULL, 0 );
    _stopWidget = XtCreateManagedWidget ( "stop",
                                          xmPushButtonWidgetClass,
                                          _w, NULL, 0 );

    // Register callbacks, specifying the object's instance
    // pointer as client data.

    addCallback ( _startWidget,
                  XmNactivateCallback,
                  &Control::start,
                  NULL );

    addCallback ( _stopWidget,
                  XmNactivateCallback,
                  &Control::stopCallback,
                  NULL );

}
```

The `stop()` and `start()` member functions are written the same as they were previously, except that the functions take additional arguments, which are unused in this example.

```
void Control::start( Widget, XtPointer, XtPointer )
{
    _timer->start();
```

```
    _stopwatch->timerStarted();
}

void Control::stop ( Widget, XtPointer, XtPointer )
{
    _timer->stop();
    _stopwatch->timerStopped();
}
```

This approach offers several advantage. It is simpler to use, given the complete implementation of the Callback class and the `Callbackdeclare` macro. It is also slightly more type-safe. The only cast occurs in association with the `clientData` argument to the `callbackStub()` function. However, this is known to be a Callback object, regardless of what derived class is actually being used, making this cast almost completely safe.

Using the `Callbackdeclare` macro reduces the number of functions a programmer must write and relieves the tedium of defining two functions for every callback. The total number of functions is also reduced, since the same static member function is used for all callbacks registered by a given class. Finally, some may find this technique more aesthetically appealing than registering static member functions and dealing with the C functions directly.

Nothing comes for free, and there are a few disadvantages to this approach. First, the Callback mechanism adds yet one more level of indirection, requiring an additional function call for each callback. This is probably not objectionable in most cases.

A second problem could be more problematic in some cases. When a widget is destroyed, Xt removes all callbacks from the widget's callback lists. When using this approach, Xt will also remove the `callbackStub()` function registered with the widget. However, it cannot know to delete the Callback object. If widgets are destroyed dynamically, the implementation described above will continue to use memory even though the functions supported by the class will never be called.

It is possible to work around this problem by registering an `XmNdestroyCallback` function for each widget and each callback class, whose task it is to free the Callback class when the widget is destroyed. However, this adds even more to the complexity and overhead of the underlying mechanisms that implement this approach.

I would like to credit Anil Pal for suggesting this approach, and demonstrating an earlier version, from which the Callback class evolved.