Encapsulating Xt Callback Functions in C++

Paul Eissen

8 November 1991*

1 Introduction

A programmer writing an application in C [1] with the X Toolkit Intrinsics (Xt) [2] typically structures the user interface component as set of *callback* functions. Each callback function is registered via Xt with an instance of a *widget* from a widget library. At runtime, user interaction (such as mouse button presses in a push button widget) cause the Xt event dispatcher to invoke registered callback functions. A callback function may be passed both widget-specific and application-specific data. Constructing an Xt-based user interface program in C, then, is an exercise in deciding on the quantity and types of widgets to be instantiated, registering callback functions with a subset of these instances, and associating application-specific data with callback functions.

Unfortunately, coding Xt-based user interface programs in C prevents a programmer from mapping the problem domain directly into an implementation. The requirements may call for, say, a radio receiver window, but the programmer must think in terms of generic widgets and domain-specific callback functions that operate on these widgets. The C++ [3] *class* provides a better model for mapping the problem domain into software. A C++ class for a radio receiver window can collect both the widget instances used to construct the actual window *and* the callback functions that operate on these instances.

This paper presents a simple technique for encapsulating Xt callback functions in C++ classes. An example C++ class will be described first. Next, the most logical (but incorrect) method for encapsulation is presented. Finally, a technique based on static member functions is described.

*Translated from the original `troff` and lightly edited on 11 February 2018.

2 An Example Class

Consider a C++ `TuneList` class that models un unbounded list of frequency values. A user can scroll through the contents of a `TuneList` window and select one frequency at a time. A `TuneList` window can manage an instance of an OSF/Motif [4] `XmList` widget. Here is the preliminary class definition:

```
#include <Xm/List.h>

class TuneList {
public:
    TuneList(const Widget parent,
             const char* name);
    virtual ~TuneList();
    ...
protected:
    ...
private:
    Widget listw;
    ...
};
```

An `XmList` widget defines several *callback lists* on which to register callback functions. We are interested in "registering" a virtual member function `frequencySelected()` on one of these lists. The most logical place for callback registration is in `TuneList`'s constructor:

```
TuneList::TuneList(const Widget parent,
                   const String name)
{
    Arg args[10];
    Cardinal n = 0;

    XtSetArg(
        args[n], XmNselectionPolicy,
        XmSINGLE_SELECT); n++;
    listw = XmCreateScrolledList(
        parent, name, args, n);
    XtManageChild(listw);

    // Register frequencySelected()?
    XtAddCallback(
        listw,
        XmNsingleSelectionCallback,
        ... );
}
```

When the user selects a frequency in a `TuneList` window, we want the runtime system to invoke `frequencySelected()` and pass it the new value.

3 The Wrong Way

At first glance, encapsulating `frequencySelected()` can be accomplished by: (1) coding it as a "regular" Xt callback function:

```
void TuneList::frequencySelected(
    Widget widget,
    XtPointer clientd,
    XtPointer calld)
{
    // Dereference "calld" to obtain
    // the new frequency
}
```

(2) adding it to `TuneList`'s `protected` list of members, and (3) registering the function with Xt:

```
XtAddCallback(listw,
    XmNsingleSelectionCallback,
    (XtCallbackProc)frequencySelected,
    (XtPointer)NULL);
```

This simple statment contains a serious type violation. Our class function *cannot* be cast to

```
void (*)(Widget, XtPointer, Xtpointer)
```

(the typedef for `XtCallbackProc`) because its type is really

```
void (TuneList::*)(
    Widget, XtPointer, XtPointer)
```

Why is this a problem?

> A pointer to member function for a particular object may be cast into a pointer to function, for example, `(int(*)())p->f`. The result is a pointer to the function that would have been called using that member function for that particular object. Any use of the resulting pointer is... undefined.[1] [3, pp. 631-632]

As a rule, non-static class member functions cannot be registered as Xt callback functions.

4 The Right Way

It is legal in C++ to pass a *static* member function pointer to `XtAddCallback()` because its type definition does *not* include a class name. We can therefore arrange to have a static member function `realSelectionCB()` invoke `frequencySelected()` through a pointer to a `TuneList` object. Since the former has no `this` pointer, we must pass it in as client data:

```
XtAddCallback(listw,
    XmNsingleSelectionCallback,
    (XtCallbackProc)realSelectionCB,
    (XtPointer)this);
```

`realSelectionCB()` then simply "forwards" pseudo-callback function invocations:

[1] As proof, two C++ compilers (g++ 1.35.1 and Sun C++ 2.1) were used to compile the `XtAddCallback()` statement. In both cases, a segmentation fault occurred when `frequencySelected()` tried to dereference its `calld` parameter.

```
void TuneList::realSelectionCB(Widget,
    XtPointer clientd,
    XtPointer calld)
{
    TuneList* p = (TuneList*)clientd;
    p->frequencySelected(calld);
}
```

`frequencySelected()`, no longer an Xt callback function, can throw away the extra parameters:

```
void TuneList::frequencySelected(
    XtPointer calld)
{
    // Perform some action on the
    // selected value in "calld"
}
```

The `TuneList` class now has the following definition:

```
class TuneList {
public:
    TuneList(const Widget parent,
             const char* name);
    virtual ~TuneList();
    ...
protected:
    virtual void frequencySelected(
        XtPointer calld);
    ...
private:
    static void realSelectionCB(
        Widget widget,
        XtPointer clientd,
        XtPointer calld);
    Widget listw;
    ...
};
```

When the user selects a frequency, Xt will invoke `realSelectionCB()`, which in turn calls `frequencySelected()` (in the context of a `TuneList` object) to perform some action on the selected value. Classes derived from `TuneList` "inherit" this method of callback invocation. A `ShortwaveTuneList` class, for example, may define its own version of `frequencySelected()`. This works because `realSelectionCB()` is *polymorphic*.

5 Conclusion

Xt callback functions in C++ are encapsulated as static class member classes. These callback functions can in turn call pseudo-callback irtual functions that do the real work. A derived class need not know (nor care) how its base class interacts with Xt. With C++ (and a little ingenuity), callback functions can be collected, protected, and inherited with ease.

References

[1] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*, volume 2nd ed. Prentice Hall, Englewood Cliffs, NJ, 1988.

[2] Paul J. Asente and Ralph R. Swick. *X Window System Toolkit: The Complete Programmer's Guide and Specification.* Digital Press, Bedford, MA, 1990.

[3] Bjarne Stroustrup. *The C++ Programming Language*, volume 2nd ed. Addison-Wesley, Reading, MA, 1991.

[4] Open Software Foundation. *OSF/Motif Programmer's Reference, Revision 1.1.* 1991.