Melding OSF/Motif, C++ and the Intrinsics

Douglas S. Rand, Gilles Benati[†]

Abstract

This paper explores alternatives and issues in developing mechanisms for integrating C++ and Intrinsics widgets in the OSF/Motif™[‡] environment. We start with a discussion of existing interfaces from C++ to user interface toolkit. Next we explain the goals and requirements for the project. We then explain our own method which allows for true Intrinsics subclasses of widgets while remaining in a C++ paradigm. The paper concludes with an example of the C++ subclassing mechanism.

Introduction

C++ and the Intrinsics are each a separate object-oriented system with its own semantics. Our goal is to make these two systems work together to provide a subclassing mechanism for C++ programmers who are using OSF/Motif. Further we wanted all the functionality of the Intrinsics to remain, and all this must require no changes to the X11 release 5 Intrinsics.

Background

There are two kinds of interfaces from C++ to a system such as the X toolkit. The first kind is a wrapper. A **wrapper** is a C++ class which contains a pointer to another object and which presents an interface to the other object.

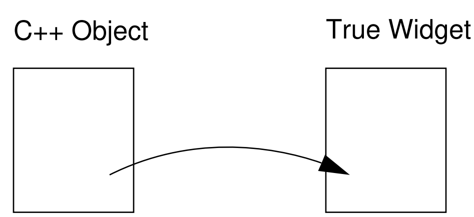

Figure 1: Wrapper Architecture

[†]*Douglas S. Rand (drand@osf.org) is a senior engineer at the Open Software Foundation, Cambridge, Massachusetts. Gilles Benati (G.Benati@frmy.bull.fr) is an engineer at Groupe Bull, Massy, France.*

[‡]*OSF/Motif and Motif are trademarks of the Open Software Foundation.*

Wrapper implementations to Motif™ include GINA++, ULowell Motif-C++, WWL and the commercial package View.h++™[†]. While wrappers can provide valuable functionality, and encapsulate behavior for a C++ programmer, they do not allow many of the more valuable features of the Intrinsics that true subclasses provide. They must also provide their own implementation of each of the functions in the Intrinsics or require the programmer to obtain the true widget to manipulate. Either a casting operator, method or both can supply direct access to the underlying widget.

Some examples of the missing functionality are:

- Use of the resource mechanism for class resources, or any resource setting for the C++ object instance[‡].
- Action procedure lists.
- Access to exposure compression and other Intrinsics performance enhancements.
- Intrinsics supported translation tables.

Supporting these are difficult since functionality already supplied as part of the Intrinsics must be duplicated.

The second kind of interface are the C++ toolkits. Object Interface™[††] is a commercial package which presents both OSF/Motif and OpenLook™[‡‡] appearances by means of a C++ class library. InterViews provides a non-commercial C++ class library. In addition, the X consortium is working on a standard IDL interface system as the next generation toolkit, Fresco, which will presumably have a C++ binding available.

These systems often provide easier paradigms for the application programmer, but since they are not based on the Intrinsics, they cannot work with Motif. This is an important requirement for us, since we wish to provide subclassing of Motif, which is based on the Intrinsics.

Requirements and Goals

We required that the object be both a C++ and an Intrinsics object. This removes the requirement to provide an entire suite of C++ interfaces to access the functionality in the Intrinsics. Unfortunately, this requirement removed the possibility of using an existing wrapper based solution, and since we wish to provide Intrinsics objects, the other existing solutions were also disqualified. Additionally, we wanted to allow a C program to utilize a C++ widget. This could be relevant, for example, in a place where one group is developing a graphical view widget in C++, but the user interface constructors are strictly working in C.

[†] *View.h++ is a trademark of Rogue Wave Software Inc.*

[‡] *This can be awkwardly handled through subresource mechanisms, not an easy option for those who are not expert widget implementors.*

[††] *Object Interface is a trademark of Solbourne Inc.*

[‡‡] *OpenLook is a trademark of AT&T.*

Our most important goals in this work are:

- To allow C++ subclassing of the Motif class hierarchy.
- To have objects which are, at the same time, both true C++ objects and Intrinsics objects.
- To limit resources used.
- To run on X11 release 5.
- To maintain highest possible degree of fidelity to C++ programming paradigm.

But the goals do not include:

- Hiding all the Intrinsics objects in C++.
- Wrapping all Motif classes in C++ wrappers.

We do not simply wish to duplicate what others have adequately done in both commercial and non-commercial offerings. The attraction of these non-goals is that they would allow the application programmer using C++ to code their entire application with only an object paradigm. So instead of calling **XtAppInitialize**, one would create a new **Application** object. Instead of calling **XtAppMainLoop**, one would call the method **Application::MainLoop**, and so on.

The cost of wrapping just the Motif class objects would be high. There are about seventy classes today, and every additional class added would need to be wrapped. Additionally, one must address the proper methodology to deal with sets of resources that should naturally be changed together. This is not generally addressed in C++ interfaces, and would require significant effort. Finally, for a true C++ like interface, at least the Intrinsics functions that deal with widgets would be interfaced as methods to the wrapper classes.

Interfacing C++ and the Intrinsics

How do we satisfy the difficult goal of making our objects both C++ and Intrinsics objects? To fulfill this the object must undergo both the initialization which the Intrinsics supply as well as that of C++.

For the object to correspond in C++ and the Intrinsics, we must exploit the correspondence of inheritance in C++ to the structure of the Intrinsics widgets. In both cases, each succeeding subclass's instance data is placed in memory. Thus we will end up with a C++ object with the form, using the example class of **StaticText**, as shown in figure 2.

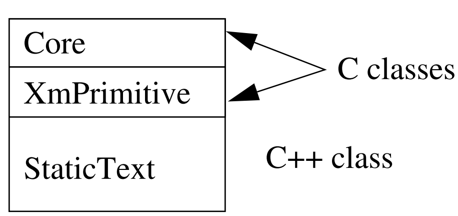

Figure 2: C++ Intrinsics Object

Classes and metaclasses

The Intrinsics have two kinds of objects defined for each widget class, class records and class instances. These are separated since C has no mechanism to handle class versus instance data. But the class record also serves as a metaclass metaphor. As a metaclass it allows the programmer to modify class creation and destruction behavior and define the side effects which occur when the object's values are manipulated through a call to the Intrinsics **Core** class method **SetValues**.

Since we need class records, we create class record objects in C++. These provide a solution to the problems of static initialization in C++ of a class record structure. The instance record becomes a C++ class as well. So we maintain the distinction of a class and metaclass for each new widget defined in C++.

In particular we have two defined pairs of classes which form the base for all subclassing which is to occur.

- **XmCxxPrimitive** (widget class) with **XmCxxPrimitiveClass** (widget class record) provide the interface to the **XmPrimitive** widget class.
- **XmCxxManager** (widget class) with **XmCxxManagerClass** (widget class record) provide the interface to the **XmManager** widget class.

These C++ base classes are abstract classes, that is they are not intended to be instantiated by the widget user. This is very much like **XmPrimitive** and **XmManager** which are generally not instantiated. The class hierarchy is shown in figure 3 of the derived classes **StaticText** and **Grid**. This also shows the general structural relationship of the C++ class hierarchy and the Intrinsics class hierarchy.

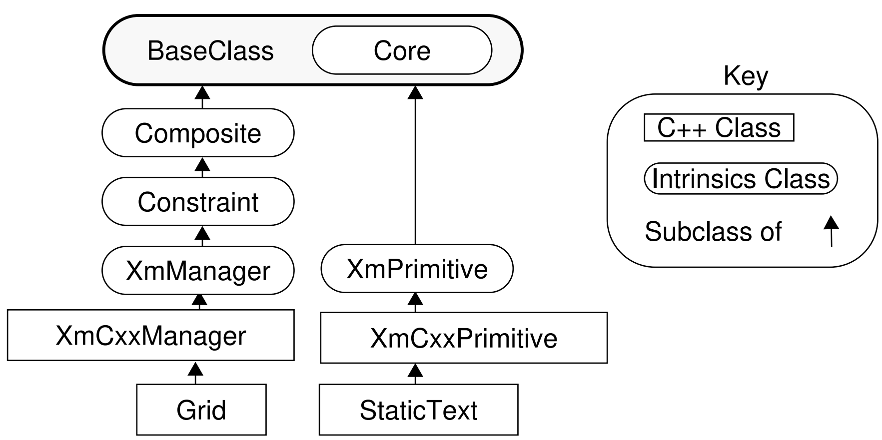

Figure 3: Class Hierarchy

Creation and destruction of objects

Both the Intrinsics and C++ wish to have control over the entire creation of their objects. In coarse detail, the Intrinsics perform roughly the following steps when a widget is created:

- First the widget class is initialized, if needed.
- The widget instance is allocated and the Constraint record is allocated for Constraint children.
- The basic information is set into the widget instance, for example parent, self, and class.
- Resources are recovered for this widget.
- Arguments to **XtCreateWidget** are applied.
- If this is a Constraint child then the constraint record is initialized.
- The **InsertChild** method is run in the Composite parent.

It is worth noting that nowhere in this list of actions is there an opportunity to allocate the object other then by the built-in invocation of **XtMalloc** during widget creation.

You may recall that one of our requirements is to run on X11 release 5. As the earliest the Intrinsics could change would be release 6, we wondered whether C++ could accommodate the release 5 Intrinsics.

C++ has a much simpler mechanism than the Intrinsics for initializing a class instance. First it runs the operator **new** to allocate storage and then it runs the constructors from superclass to subclass. Some control is given to the class designer and programmer in how they pass arguments down the constructor chain to the superclass, and what private protocols are defined by the class designer.

Virtual functions in C++

In C++ there is the ability to have run-time dispatching of member functions depending on the type of the instance operated on. These are known as virtual functions. Each class has a table of the virtual functions which can be used with a pointer to the proper function. As an example, imagine that we had a family of classes whose parent class was Shape. Each subclass (Circle, Square) of Shape defines a function **Draw** which renders the particular subclass. Now we can have a pointer to an instance of Shape and invoke the **Draw** method and the correct method will run.

The C++ operator new

In C++ one creates objects using the operator **new** and removes the objects with operator **delete**. These operators are also used for dynamic allocation of built-in objects such as floating point numbers, integers, character arrays, and structures, and generally removes the need for directly calling **malloc** and **free**. For class objects, one also can specify arguments to the *constructor*. The constructor is available in a number of forms. So an instance of class car might be constructed as follows:

```
Car *mycar, *yourcar, *anothercar;

mycar = new Car();               // Default constructor
yourcar = new Car(10,10);        // Constructor taking some arguments
anothercar = new Car(yourcar);   // Copying constructor
```

The C++ operator **new**, the allocation operator, can be overloaded to take arguments. By default, **new** takes a single argument which is the size of the argument to be created. Operator **new** returns a pointer to the allocated memory, which is then initialized with all static data for that C++ class. The size argument is automatically supplied by the compiler.

Most importantly, the **new** operator initializes the virtual function table for the instance. So we could take an object of class **Bar** and make it into a class **Foo** object by the method in figure 4.

```
operator new(size_t ignored, void* pointer)
   { return(pointer); }

oldobject = new Bar();                        // Initial creation of object
reinitialized_object = new(oldobject) Foo(); // Recreation of object
```

Figure 4: Changing the class of an object in C++

As the **new** operator can be overloaded with other, more extended argument lists, one can pass in a pointer to an existing object which has been allocated. So by defining a version of operator **new** which takes two arguments, the size and a pointer to an object, one can ignore the size and return the pointer. This returned pointer is then operated on, just as in the default case, resulting in an initialized C++ object of the specified class.

The Motif BaseClass Intrinsics class

Motif modifies the Intrinsics **Core** class record to include more semantics for most all aspects of widget creation and modification. The modifications to the **Core** class are known as **BaseClass**. This wraps most of the basic **Core** methods to allow for more flexible initialization and SetValues semantics.

One particular class method in **BaseClass** is known as **InitializePrehook**. This is only called on the current class, i.e. it is not chained, and is called before the **Initialize** methods are chained. It is an ideal place to use operator **new** to initialize the C++ instance. This could have also been done in **Initialize** with some more trouble, as **Initialize** is called for all superclasses, and we would prefer to only call operator **new** once.

This will result in the Intrinsics object having the correct virtual table installed for C++. This is crucial since we rely on the virtual function mechanism in C++ to provide the run-time dispatching we need. It is mandatory that each subclass defines a similar **InitializePrehook** method. An example of what the this method looks like for the C++ subclass **Grid** is shown in figure 11 later in this paper.

To construct a new object the user then uses **XtCreateWidget**, just as if the subclass were a normal C subclass. When the **BaseClass** initialization calls the **InitializePrehook** class method, the object becomes a real C++ object in the “new in place” call.

It is worth mentioning that we considered defining operator new with the additional arguments required by **XtCreateWidget** and calling **XtCreateWidget** within the operator. This is more C++ like, but since many or most of the widgets will still be created by using **XtCreateWidget**, we did not feel this was a significant advantage. There was also the disadvantage of making it more difficult for a C programmer to use a widget created in this way.

Intrinsics and C++ destruction models

The Intrinsics perform roughly the following steps when deleting a widget:

- The destroy callbacks are called.
- The Composite parent’s **DeleteChild** method is called.
- The **Destroy** methods are invoked in subclass to superclass order.
- The window is destroyed and the widget’s memory is freed.

The destruction model in C++ is much simpler. When a C++ instance is destroyed, all the destructors are called in subclass to superclass order. Any other processing must be done in the destructors themselves. Importantly, the operator which invokes the destructors, operator **delete**, may be virtual, which allows the correct destructors to be called automatically.

Destruction of C++ Intrinsics subclasses

An instance is easily removed with the **delete** operator. This operator causes the *destructors* for the instance to be run. If the operator **delete** is declared *virtual*, then an instance of unknown class[†] will have the proper destructors run. For the non-virtual **delete** operator, the compiler uses the type of the pointer to indicate which class’s destructors will be run.

The virtual delete operator thus gave us an entirely sufficient replacement for the **Destroy** method in the Intrinsics Core class. So for the C++ classes, the destructors have the responsibility to free allocated storage which is normally done in the **Destroy** method.

We tie the C++ and Intrinsics models together by having the abstract class’s **Destroy** Intrinsics class method invoke the C++ operator **delete**, which we define as a virtual deletion operator. This simple technique assures that all subclasses will have their destructors called correctly. The C++ programmer can then put the necessary steps in the destructor and not bother with the Intrinsics **Destroy** method.

Class Record Classes

When someone subclasses in C, they normally create a static record structure which becomes the widget class. This static structure includes all the data which the Intrinsics need to create the instances of the widget class such as the size of the instance, the translation tables, pointers to

---

[†] *Unknown class, but of a known base class.*

initialization functions for the class and instance, and pointers to the class methods for this class (such as **Resize**, **Expose**, **SetValues**, etc.).

We wanted to simplify this mechanism as much as we could, and provide a mechanism that fits the C++ paradigm. We settled on defining class record classes in C++ which correspond to the Motif widget set classes of **XmPrimitive** and **XmManager**.

Constructor

We only need to have arguments to the constructor of items in the class record which are not handled by the C++ interface. The constructor signature for the **XmCxxPrimitiveClass** is in figure 5, that of **XmCxxManagerClass** is identical.

```
XmCxxPrimitiveClass(char*             name,
                    WidgetClass       parent,
                    Cardinal          widgetSize,
                    XtProc            class_init,
                    XtActionList      actions,
                    Cardinal          num_actions,
                    XtResourceList    resources,
                    Cardinal          num_resources,
                    XtInitProc        cxx_cast);
```

Figure 5: XmCxxPrimitiveClass's Constructor Signature

A number of values are defaulted in the constructor, such as the compression parameters in **Core**, but these can be overridden in the classInitialize procedure. We also setup the class methods which map into virtual functions in this constructor. **XmCxxManagerClass** is very much like **XmCxxPrimitiveClass** with some additions.

While the constructor above has many arguments, one need only look into the Intrinsics source code to realize that this is but a fraction of the arguments supplied in a **Core** subclass. Further, if one creates a subclass of **XmCxxPrimitiveClass**, its constructor need only include the arguments for its subclasses, so instead of an ever growing list of items to fill into the class record, one need only add the new metaclass methods. There will normally be none, since most such functionality is more easily done via the C++ virtual function mechanism.

Resource lists

One of the main reasons to have a native C++ implementation of a widget is to take advantage of the Intrinsics resource mechanism. The resource list contains the name, type and offset of the resources in the instance structure. This instance structure must be contiguous. Dealing with resources for external structures is much more difficult.

This was an area which we were unable to significantly make easier than the same task in the Intrinsics. We require the subclasser to perform the same steps which are taken in standard Intrinsics subclassing. In fact, all answers we found seemed to add and not subtract from the task.

Core class methods

The Intrinsics class methods on **Core** may be thought of in two groups. The first group are methods which apply to an individual widget and are specialized for a particular widget class. These include methods such as **Expose** and **Resize**. The second group are those methods which define class behavior, or what may be termed metaclass methods.

Virtual function translations

Most of the class methods which apply to widgets are translated by class methods in the foundry classes to virtual function calls. The most important examples of this are the methods for **Expose** and **Resize** which become calls to virtual function. C++ subclasses then have a much simpler mechanism for overriding, inheriting or calling a parent class's method.

The complete definition of **XmCxxPrimitive** follows. Note the very large percentage of all the methods which we were able to make into virtual functions.

```
class XmCxxPrimitive {
   friend class XmCxxPrimitiveClass;
public:
   CorePart          core;
   XmPrimitivePart   primitive;
protected:
   // constructors & destructors
   void* operator    new(size_t size, void* w = NULL);
   void  operator    delete(void*);
   virtual ~XmCxxPrimitive();
   XmCxxPrimitive();
   // virtual chained class-methods
   virtual void      initialize(Widget req_w, ArgList, Cardinal*);
   virtual void      initialize_hook(ArgList, Cardinal*);
   virtual Boolean   set_values(Widget old_w, Widget req_w,
                                ArgList, Cardinal*);
   virtual Boolean   set_values_hook(ArgList, Cardinal*);
   virtual void      get_values_hook(ArgList, Cardinal*);
   // virtual self contained class-methods
   virtual void      realize(XtValueMask* mask,
                             XSetWindowAttributes* win_att);
   virtual void      resize();
   virtual void      expose(XEvent*, Region);
   virtual Boolean   accept_focus(Time* t);
   virtual void      set_values_almost(Widget old_w,
                                       XtWidgetGeometry* request,
                                       XtWidgetGeometry* reply);
   virtual XtGeometryResult
                     query_geometry(XtWidgetGeometry* intended,
                                    XtWidgetGeometry* reply);
   // base class extension part
   virtual Boolean   set_values_prehook(Widget old_w, Widget req_w,
                                        ArgList, Cardinal*);
   virtual void      initialize_posthook(Widget req_w,ArgList, Cardinal*);
   virtual Boolean   set_values_posthook(Widget old_w, Widget req_w,
                                         ArgList, Cardinal*);
   virtual void      secondary_object_create(Widget req_w,ArgList,
                                             Cardinal*);
   virtual void      get_values_prehook(ArgList, Cardinal*);
   virtual void      get_values_posthook(ArgList, Cardinal*);
```

```
   virtual XmNavigability  widget_navigable();
   virtual void      focus_change(XmFocusChange);
   virtual void      display_accelerator(String);
   // primitive class part
   virtual void      border_highlight();
   virtual void      border_unhighlight();
   virtual void      arm_and_activate(XEvent* event,String* params,
                                      Cardinal* nparams);
   // primitive extension part
   virtual Boolean   widget_baseline(Dimension** dim, int* nb);
   virtual Boolean   widget_display_rect(XRectangle*);
   virtual void      widget_margins(XmBaselineMargins*);
public:
   // static class-methods
   // must be public : passed as parameter to class_rec static constructor
   static void       _MakeCxxWidget(Widget req, Widget new_w,
                                   ArgList, Cardinal*);
};
```

Additionally the following is extracted from the definition of **XmCxxManager**. This shows only the methods peculiar to **XmManager**, the **Core** methods have been omitted for brevity.

```
class XmCxxManager {
   friend class XmCxxManagerClass;
public:
   // convenience functions
   Widget      Child(int index) { return composite.children[index];}
   Widget      LastChild() {
         return composite.children[composite.num_children -1];
   }
   CorePart          core;
   CompositePart     composite;
   ConstraintPart    constraint;
   XmManagerPart     manager;
   // composite part
   virtual XtGeometryResult geometry_manager(Widget,
                                             XtWidgetGeometry*,
                                             XtWidgetGeometry*);
   virtual void      change_managed();
   virtual void      insert_child(Widget);
   virtual void      delete_child(Widget);
   // manager part
   virtual Boolean   parent_process(XmParentProcessData);
   // manager extension part
   virtual Boolean   traversal_children(Widget**, Cardinal*);
   // manager class methods
   static Boolean    ParentProcess(Widget, XmParentProcessData);
   // manager extension class methods
   static Boolean    TraversalChildren(Widget, Widget **, Cardinal *);
};
```

Metaclass methods and Chained methods

ClassInitialize couldn't use the virtual function mechanism. This is not called with a widget and thus cannot dispatch to a specific function. Additionally, a number of the regular class methods which the Intrinsics chain automatically and there was an issue of whether these should use virtual functions or not.

The list of functions that the Intrinsics chain automatically is not long. Primarily these are metaclass functions which can become virtual because they have arguments to dispatch on. The most notable such function is **SetValues**.

We debated whether to have these chained methods be virtual or not. If they would be virtual, then we would need to assure that they are only called once per invocation of the Intrinsics class method, and secondarily we would need to assure that the virtual functions were properly chained. The only mechanism to assure that the functions would be properly chained would be to document the requirement, no automatic mechanism would work.

On the other hand, there are distinct advantages to converting the chained methods into virtual functions with manual chaining. First, we would create a uniform system where all the class based operations on the widgets are virtual. Second, the programmer could then choose when to do the method combination. Currently, Motif has provided a simpler version of this in the **BaseClass** extensions by creating hooks, and the Intrinsics have some specific hooks as well, for much the same reasons.

The final decision to this debate appears to have gone to the virtual function route. The responsibility of performing the chaining goes to the widget programmer. We determined the correct answer by trying it both ways and evaluating the results. Less clear to us is whether this should have been done in the Intrinsics instead of automatic chaining.

Examples

An XmCxxManager subclass: the Grid widget

Grid is a rather simple manager which displays its children in a two dimensional table. All children are configured to have the same size. **Grid** has a few resources to control its layout (rows, columns, margin width, margin height, shadow type). It defines constraints for its children (grid margin width, grid margin height). **Grid** also defines new class methods. We will see what we gain by doing this in C++ as opposed to C.

The class record definition, shown in figure 6, is very simple. Unlike a C subclass, there is little or no “boiler plate” to write in subclassing.

```
class XmCxxGridClass : public XmCxxManagerClass {
   friend class XmCxxGrid;
public:
   XmCxxGridClass(char*          name,
                  WidgetClass    parent,
                  Cardinal       widgetSize,
                  XtProc         class_init,
                  XtActionList   actions,
                  Cardinal       num_actions,
                  XtResourceList resources,
                  Cardinal       num_resources,
                  XtInitProc     cxx_cast);
   // static class-methods
   // must be public : passed as parameter to class_rec static constructor
   static void       ClassInitialize();
protected:
   XtPointer cxx_grid_extension;
};
```

Figure 6: GridClass definition

Next, in figure 7, we see the definition of the class.

```
class XmCxxGrid : public XmCxxManager {
   friend class XmCxxGridClass;
public:
   // should really be protected but we must reference fields when
   // declaring the static resources array
   XmCxxGridPart grid;
protected:
   // Here is a list of 'file-static' functions (i.e. meant for the widget
   // programmer) that could be used by a subclass, so make them protected
   // rather than private.
   void              DrawShadow();
   void              ClearShadow();
   void              ConfigureChildren(Widget instigator,
                                       XtWidgetGeometry *desired);
   void              CalcGridSize(Widget child,
                                  Dimension childWidth,
                                  Dimension childHeight,
                                  Dimension childBW,
```

```
                              Dimension *gridWidth,
                              Dimension *gridHeight);
   // virtual chained class-methods
   // core class part
   virtual void      initialize(Widget req_w, ArgList, Cardinal*);
   virtual Boolean   set_values(Widget old_w, Widget req_w,
                                ArgList, Cardinal*);
   // constraint class part
   virtual void      constraint_initialize(Widget req_w,
                                          Widget new_w,
                                          ArgList, Cardinal*);
   virtual void      constraint_destroy(Widget w);
   virtual Boolean   constraint_set_values(Widget old_w, Widget req_w,
                                          Widget new_w,ArgList, Cardinal*);
   // virtual self contained class-methods
   // core class part
   virtual void      resize();
   virtual void      expose(XEvent*, Region);
   virtual XtGeometryResult
                     query_geometry(XtWidgetGeometry *intended,
                                    XtWidgetGeometry *reply);
   // composite class part
   virtual XtGeometryResult
                     geometry_manager(Widget child,
                                      XtWidgetGeometry* intended,
                                      XtWidgetGeometry* reply);
   virtual void      change_managed();
public:
   // static class-methods
   // must be public : passed as parameter to class_rec static constructor
   static void       _MakeCxxWidget(Widget req, Widget new_w,
                                    ArgList, Cardinal*);
};
```

Figure 7: Grid definition

Defining new class methods for a C++ class is simple. Instead of defining new class methods in the class record, one need only add a new virtual member function to the C++ instance class. They will be stored in the virtual function table by the compiler and inheritance will be done automatically by C++.

We would like to hide more data and methods by making them protected or private. Unfortunately this is not always possible. In particular:

- The fields used to store the resources are referenced in the static resource array declaration, thus they have to be public.
- The static Xt class methods are given as parameters to the static class record constructor, so they must be public as well.

```
// constraint record
struct XmCxxGridConstraintPart {
     Dimension grid_margin_width;
     Dimension grid_margin_height;
};
```

```
struct XmCxxGridConstraintRec {
     XmManagerConstraintPart manager;
     XmCxxGridConstraintPart grid;
};
typedef XmCxxGridConstraintRec* XmCxxGridConstraint;
```

Figure 8: Grid constraint definition

Note that the constraint records are just declared as structures. It would be possible, but not worthwhile to make these C++ objects.

```
XmCxxGridClass::XmCxxGridClass(/* long parameter list*/)
        : XmCxxManagerClass(/* long parameter list */) {
   // core part
   core_class.compress_motion          = TRUE;
   core_class.compress_exposure        = XtExposeCompressMaximal;
   core_class.compress_enterleave      = TRUE;
   // constraint_class fields
   constraint_class.resources          = constraints;
   constraint_class.num_resources      = XtNumber(constraints);
   constraint_class.constraint_size    = sizeof(GridConstraintRec);
   constraint_class.initialize         = Grid::ConstraintInitialize;
   constraint_class.destroy            = Grid::ConstraintDestroy;
   constraint_class.set_values         = Grid::ConstraintSetValues;
   // manager_class fields
   manager_class.translations          = NULL;
   manager_class.syn_resources         = syn_resources;
   manager_class.num_syn_resources     = XtNumber(syn_resources);
   manager_class.syn_constraint_resources    = syn_constraints;
   manager_class.num_syn_constraint_resources= XtNumber(syn_constraints);
   // grid class fields;
   grid_class_extension                = NULL;
}
```

Figure 9: GridClass constructor

Most of the methods from **Composite** class and **Core** class are inherited. Because these functions are chained by the Intrinsics, one has to explicitly set the constraint class methods. A subclass of **Grid** would have to override them either by NULL or by its own functions. Note that as with other class methods chained by the Intrinsics, these have the same problems as talked about before in "Non-virtual methods." The management of synthetic resources, constraints and synthetic constraints suffer the same shortcoming in that each subclass must override them or set them to NULL.

```
// static instance of the class record.
XmCxxGridClass XmCxxGridClassRec("XmCxxGrid",
                        &XmCxxManagerClassRec,
                        sizeof(XmCxxGrid),
                        XmCxxGrid::ClassInitialize,
                        NULL,     // actions
                        0,        // numActions
                        resources,
                        XtNumber(resources),
                        XmCxxGrid::_MakeCxxWidget);
```

```
WidgetClass XmCxxGridWidgetClass = (WidgetClass) &xmCxxGridClassRec;
```

Figure 10: GridClass record instance

```
void XmCxxGrid::_MakeCxxWidget(Widget req, Widget new_w, ArgList args,
                              Cardinal *num_args) {
   XmCxxGridWidget dummy = new(new_w) XmCxxGrid;
}
```

Figure 11: Grid InitializePrehook Definition

```
void XmCxxGrid::ClassInitialize() {
   XmCxxGridClassRec.SetBaseClassExtensionQuark();
}
```

Figure 12: Grid class Initialization

The **ClassPartInitialize** method is the place where the C widgets have to deal with class method inheritance for its class methods. Here, there is nothing to do since the C++ compiler takes care of it for us through the virtual function mechanism.

```
void XmCxxGrid::resize() {
   clear_shadow();
   configure_children(NULL, NULL);
   draw_shadow();
}
void XmCxxGrid::expose(XEvent* event, Region region) {
   draw_shadow();
   _XmRedisplayGadgets((Widget) this, event, region);
}
```

Figure 13: Examples of Grid virtual class methods

A minimal C++ widget

This section describes what the required steps are to create a new C++ subclass of an existing C++ widget. For the sake of argument, let's say that we want a **Grid** widget with a new **configure_children** class method. This new widget class will be called **MyGrid**.

The widget itself

```
class MyGrid : public XmCxxGrid {
   friend class MyGridClass;
public:
   // Xt class methods
   static void       ClassInitialize();
protected:
   ~MyGrid() {}
   MyGrid() {}
   // Define virtual function.
   virtual void      configure_children(Widget,XtWidgetGeometry*);
```

```
};
```

The class record

We define a new class record, a subclass of **GridClass**.

```
class MyGridClass : public XmCxxGridClass {
   friend class MyGrid;
protected:
   XtPointer mygrid_class_extension;
};
```

Note that we don't even need a constructor here, the superclass constructor is fine.

```
// static instance of the class record.
MyGridClass myGridClassRec("MyGrid",
                           &XmCxxGridClassRec,
                           sizeof(MyGrid),
                           MyGrid::ClassInitialize,
                           NULL,        // actions
                           0,           // numActions
                           NULL,        // resources
                           0,           // numResources
                           MyGrid::_MakeCxxWidget);
WidgetClass MyGridWidgetClass = (WidgetClass) &MyGridClassRec;
```

It is mandatory to define a new class record class for each new C++ widget class.

Crucial function definitions

```
void MyGrid::_MakeCxxWidget(Widget req, Widget new_w, ArgList args,
                           Cardinal *num_args) {
   MyGridWidget gw = new(new_w) MyGrid;
}
```

The casting function must be defined for every class to provide the C++/Intrinsics initialization.

```
void MyGrid::ClassInitialize() {
   myGridClassRec.SetBaseClassExtensionQuark();
}
```

Setting the **XmBaseClassExtension** quark is needed in every Motif widget, we have provided the **SetBaseClassExtensionQuark** member function to simplify this process. Most widget classes will need to do no more than call this member function.

Conclusions

Interfacing C++ and the Intrinsics turned out to be possible, but no simple task. The work has been successful, but not without its rough spots. Our examples lead us to believe that this work will prove useful to the general community of Motif users who are using C++. We hope it will encourage them to create new widgets in C++ for use in their applications.

While we believe our techniques will be portable to most C++ environments, and we have tested our code with both gcc 2.x and one flavor of cfront 2.1, we believe that portability of code may become an issue as we ship the subclassing support.

The method outlined here turned out surprisingly well. A large percentage of the Intrinsics class methods were amenable to becoming virtual functions, making them easy for a C++ programmer to combine, and override. The exceptions were class behavior methods in the Intrinsics, many of which take no argument.

But, how well have we met our goals? We feel comfortable in stating that most of our goals were entirely met. The single exception is the degree to which we could remain in a strictly C++ paradigm. Here we had to make use of the precise Intrinsics mechanism in providing the metaclass methods. Another area which was not tractable was our ability to hide the need for knowledge of the Intrinsics from the subclasser. In fact, we could and did make it more palatable, but we could not remove knowledge of how the Intrinsics work.

While C++ has weaknesses, notably a fairly simple creation and destruction model in comparison with other object-oriented languages, it proved flexible enough to allow us to create native C++ Intrinsics widgets. These combination objects can use the best of each of the two worlds to accomplish their needs.

Table 1: Code size in source lines of code

Module	C	C++
StaticText	1071	576
StaticText headers	101	75
Command	550	227
Command headers	68	53

Of interest for some is the difference in code size between an equivalent C and C++ widget implementation, table 1 provides the information for two of the example widgets. Static text has been discussed, and the command widget is a simple push button widget.

Acknowledgments

This work has been done as part of the Motif development effort, and as such the entire development team has been involved with aspects of the discussion. Ellis Cohen is responsible for getting the effort started and many good ideas. Credit for informing us of the overloaded new operator goes to Mark Grossman from Bellcore.

Bibliography

GINA++, The Generic Interactive Application for C++ and OSF/Motif, User Manual, Robert Babatz, Andreas Bäcker, GMD, 1991.

WWL, a Widget Wrapper Library for C++, version 1.2, Jean-Daniel Fekete, Laboratoire de Recherche en Informatique, Orsay Cedex, France, 1991.

A C++ Binding for OSF/Motif, K. Seetharaman, C. F. Rei, B. R. Montagne, University of Lowell, Lowell, MA, 1990.

OI Programmer's Guide, A. Benson, G. Aitken, Prentice Hall, Englewood Cliffs, NJ, 1992.

InterViews: A C++ Graphical Interface Toolkit, M. A. Linton, et al, Stanford University, 1987.

X Window System Toolkit, The Complete Programmer's Guide and Specification, P. Asente and R. Swick, Digital Press, 1990.

The C++ Programming Language, Second Edition, B. Stroustrup, Addison-Wesley, 1991.