Chapter 30

Custom Controls and Handlers

- Describes how to create your own custom control and handler classes to extend the function of the User Interface Library.
- Creates a value-set control class and associated event handler classes as an example.
- You should read Chapters 4 and 7 before reading this chapter.
- Chapters 15 and 17 cover related material.

This chapter describes how you can extend the User Interface Library by writing your own window classes to encapsulate custom controls, and by writing handler classes to process input and notification events for your custom control windows.

To illustrate how you accomplish these tasks, we present a full description of the Presentation Manager value set control as a set of C++ classes.

Custom Controls

We use the term *custom controls* to refer to graphical user interface components that provide some fundamentally different appearance or user input capability.

This section will describe how you create your own window classes that encapsulate custom controls. These classes will be added to the User Interface Library's `IWindow` hierarchy, most likely as a class derived from `IControl`. Sometimes, you will derive from a more specialized control class already present in the class hierarchy beneath `IControl`.

What we will cover in this chapter are the special techniques you need to know to encapsulate new, and improved, user interface components. The objective is to make your new classes fit into the User Interface Library design. This will make your objects easier to use by others, and enable them to integrate with the rest of the User Interface Library features.

Placing Your Control in the IControl Hierarchy

The first step toward implementing a C++ custom control is to determine where to place your control in the `IWindow` hierarchy. In most cases, your new class will be derived from `IControl` class.

Deciding where in the hierarchy to place your custom control class is usually easy to do. You will usually be dealing with one of the following situations.

- Encapsulating a new Presentation Manager window class.
- Implementation a custom control purely in C++.
- Extending an existing C++ window class.

New Presentation Manager Window Class

You can encapsulate new Presentation Manager window classes. As an example, in this chapter, we will be using a wrapper for the value set window class, which falls into this category. In cases such as this, you should derive from `IControl` class.

This decision follows from the fact that we are dealing with a distinct Presentation Manager window class. This window class is unlikely to provide support for the same set of Presentation Manager events and notifications, as another window class. As a result, it would prove difficult to derive from a specialized `IControl` class, which was implemented to work with some other type of Presentation Manager window.

However, you may choose to place your custom control class under one of the abstract classes derived from `IControl`, such as `ITextControl`, or `IButton`. You would do this, if your control could support the behavior required of those classes. For example, if your custom control dealt with text contents, and it made sense for an application programmer to want to apply `text` and `setText` functions to your objects, then you could derive from `ITextControl`.

Custom Control Purely in C++

You can implement custom controls purely in C++. Most likely, you will derive straight from `IControl` in these cases, too, for much the same reasons as outlined above.

In other words, you do not have a special Presentation Manager window class with a corresponding window procedure. Instead, you have implemented your user interface component using a more generic Presentation Manager window, and a set of specialized handlers. The User Interface Library's `ICanvas` class, and its derivatives, are good examples of this kind of custom control. The date control example—which uses a multicell canvas with spin buttons—given in Chapter 11, "List Controls," provides another example of this kind of custom control.

Extending an Existing C++ Window Class

You can provide a C++ encapsulation of some specialization of a Presentation Manager window already encapsulated in C++. In this case, you should derive from the existing C++ class.

A good example of this kind of extension—that technically is not a custom control—is the User Interface Library class `IGraphicPushButton`. This class uses an underlying `WC_BUTTON`, just like `IPushButton`, but provides additional C++ functionality (including a specialized owner draw handler) to make it easier for you to use the `WC_BUTTON` with graphics instead of text.

Constructing Your Custom Control Window

Naturally, the first aspect of your custom control's design we will address is construction. Your custom control has to meet two obligations during object construction.

- Ensure that a corresponding Presentation Manager window is created and attached to the C++ object representing the control.
- Properly initialize any object state specific to your custom control C++ object.

The User Interface Library controls almost always assumes that the corresponding Presentation Manager window has been constructed at the completion of the C++ object's constructor. This ensures that other member functions can be called and usually executed properly, without requiring you to call some other member function first. This convention is simply in keeping with the basic premise of C++ constructors: they ensure that the object is always in a stable and operational state. So, your custom control's constructors must take care of creating the proper Presentation Manager window.

Of course, you also will have to take care of initializing the aspects of your objects that are specific to your custom control. This will include constructing the data members of your derived class, and performing what functions are necessary, to properly initialize the state of your object and its components.

By convention, the User Interface Library controls all provide three basic constructors. These handle the three most common ways in which you can create and attach the C++ objects, and the corresponding Presentation Manager window.

Constructor With Window Handler

C++ custom controls, which correspond to Presentation Manager control windows, should provide a constructor that takes the handle of an existing Presentation Manager window. This permits you to use C++ controls created by non-C++ (or at least non-User Interface Library) programs.

If you implement your control entirely in C++, rather than encapsulating some standard Presentation Manager window class, then you probably will not provide this constructor. It is not necessary, in that case, as any window handle for one of your controls would already have a

C++ object associated with it. Remember, you cannot have two C++ objects associated with the same Presentation Manager window (for details on this, see Chapter 4, “Windows, Handlers, and Events”).

The implementation of this standard constructor should:

- Validate the input window handle.
- Ensure that the window is of the proper type.
- Attach the C++ object to the Presentation Manager window.
- Attach any special handlers that your control requires.

Below is the implementation of this constructor for our `ValueSet` example. We factor out the actual initialization code from the constructor so that we can reuse it to implement another `ValueSet` constructor.

```
ValueSet :: ValueSet ( const IWindowHandle &hwnd )
  : IControl()
  {
  this -> initialize( hwnd );
  }

void ValueSet :: initialize ( const IWindowHandle &hwnd )
  {
  IASSERTPARM( hwnd.isValid() );
  IString
    className( "#nn" );
  WinQueryClassName( hwnd, 4, className );
  IASSERTSTATE( className == "#39" );
  this -> startHandlingEventsFor( hwnd );
  this -> setAutoDestroyWindow( false );
  }
```

The call to `IWindow::startHandlingEventsFor` attaches this C++ window object to the Presentation Manager window. This causes the User Interface Library to subclass the window procedure, and enables User Interface Library handlers to be attached to the control. Window events will then be dispatched to those handlers. The User Interface Library requires you to call that function sometime before completion of your window construction process. It is necessary to enable the User Interface Library to begin intercepting window events for the control, and to route events to any handlers you have attached to the control. You can read more about this basic mechanism in Chapter 4, “Windows, Handlers, and Events.”

Notice the call to disable the automatic destruction of the Presentation Manager window, when the C++ object is destructed. Since the C++ object did not create the window, the object should not automatically destroy it. It is likely there is also non-C++ code that will take care of destroying the window. You should always disable the automatic window destruction in this standard constructor.

Constructor to Support Dialog Template

Classes that implement custom controls, which encapsulate Presentation Manager window classes should also provide a constructor that accepts as arguments a parent window pointer and a control identifier. Users will use this constructor to wrapper instances of these Presentation Manager control windows they create implicitly as components of frame windows defined by dialog template resources.

For example, you might use the dialog box editor to create a dialog template that looks like:

```
DLGTEMPLATE IC_DEFAULT_FRAME_ID
  BEGIN
    DIALOG "MyDialog", 1, 10, 10, 200, 200
      BEGIN
        VALUESET VALUESET_ID, 10, 10, 100, 100, VS_ICON
      END
  END
```

The C++ code that loads this dialog and accessed the value set would then look like:

```
IFrameWindow  dialog( IC_DEFAULT_FRAME_ID );
ValueSet  valueSet = ValueSet( &dialog, VALUESET_ID );
```

The requirements of this standard constructor are the same as for the constructor, which accepts a window handle. We can use the Presentation Manager function `WinWindowFromID` to get the window handle of the argument window. It is easy to implement this constructor by calling the same initialize function, passing the window handle obtained from that function.

```
ValueSet :: ValueSet ( IWindow *parent, unsigned long id )
  : IControl()
  {
    this -> initialize( WinWindowFromID
                          ( parent->handle(), id ) );
  }
```

As you can see, your provide this constructor mainly as a convenience and to enable usage consistent with the built-in User Interface Library controls.

Constructor From Window Attributes

The third standard constructor all `IControl` classes provide is one that takes the full set of standard control attributes.

- Control identifier
- Parent window
- Owner window
- Initial size and position
- Style

This constructor is the one that actually creates a Presentation Manager window, rather than simply attach a C++ object to an existing one. The arguments provide the information required for creation of that window.

For convenience, the last two arguments have default values. A control's default size and position is a default rectangle: positioned at (0,0) with size (0,0). The default style is the default style for the custom control class. We cover this topic later in this chapter in the section on "Custom Control Styles."

Below is the implementation of this constructor for our `ValueSet` example.

```
ValueSet :: ValueSet ( unsigned long     id,
                       IWindow          *parent,
                       IWindow          *owner,
                       const IRectangle &initial,
                       const Style      &style )
  : IControl()
  {
  this -> initialize ( ISize(0,0),
                       id,
                       parent,
                       owner,
                       initial,
                       style );
  }

void ValueSet :: initialize ( const ISize      &size,
                              unsigned long     id,
                              IWindow          *parent,
                              IWindow          *owner,
                              const IRectangle &initial,
                              const Style      &style )
  {
  IString
    ctlData = ValueSet::controlData( size );
  IWindowHandle
    hwnd = this -> create( id,
                           0,
                           style.asUnsignedLong(),
                           WC_VALUESET,
                           parent->handle(),
                           owner ? owner->handle()
                                 : IWindowHandle(0),
                           initial,
                           (char*)ctlData,
                           0 );
  this -> startHandlingEventsFor( hwnd );
  }

IString ValueSet :: controlData ( const ISize &size )
  {
  IString
    result( 0, sizeof( VSCDATA ) );
  VSCDATA
   *p = (VSCDATA*)(char*)result;
  p->cbSize = sizeof( VSCDATA );
  p->usRowCount = size.height();
  p->usColumnCount = size.width();
  return result;
  }
```

There are a few details in this code you should note. You will have to deal with the same issues when implementing your own custom control classes.

As we did with the constructors discussed previously, we have separated the construction logic from the constructor. We have placed it in another overloaded version of the `initialize` member function. This version of `initialize` function accepts an `ISize` argument that provides the number of rows and columns in the value set. This standard constructor creates value sets with zero rows and zero columns. In your custom controls, you will have to come up with similar default values for your control's attributes. Sometimes it makes sense to let the user of your class control these defaults by maintaining them as static data members of your class. You also would provide a static member function to set the default. However, this is not really useful for this characteristic of value sets. It is not likely the user of this class will have some standard size they want for all their value sets. Instead, we provide a special constructor we describe below.

Some Presentation Manager windows require you pass certain control data at creation time. With `WC_VALUESET`, we need a `VSCDATA` structure with the number of rows and columns. The static member function `controlData` creates this structure and returns it as an `IString`. We pass the address of this structure on the call to `IWindow::create`.

We connect the C++ and Presentation Manager window by calling the `IWindow` function `startHandlingEventsFor`, passing the handle of the newly created window.

Other Constructors

In addition to the standard constructors, your custom control class should provide whatever other constructors you need, to give users of your class the ability to easily create the objects of your class they will be using most often.

The number and nature of those constructors will depend on the nature of your custom control class. Generally, you will provide constructors that let users of your class pass in the data corresponding to the information the Presentation Manager window class accepts, or requires, in the control data argument on the `WinCreateWindow` call. If there is no control data, or if there is some reasonable default you can always provide, then you will not need any additional constructors.

In the case of `ValueSet`, we see the `WC_VALUESET` control data structure has fields for the number of rows, and number of columns, in the value set. In the standard constructor, described above, we provided 0 as the default values for these numbers. Of course, a value set with no rows and no columns is not very useful, so we have an obligation to provide a constructor, which lets the users of our class provide the actual number of rows and columns they need.

We use an `ISize` argument to pass the value set size. This works well since it provides the right "size" connotation and permits the information to be passed easily. We insert this `ISize` argument before the arguments that have default values, so the user can still rely on those defaults even if they want to provide an explicit size. This yields the following `ValueSet` constructor. We use the initialize member function we discussed above, to implement this constructor:

```
ValueSet :: ValueSet ( unsigned long     id,
                       IWindow          *parent,
                       IWindow          *owner,
                       const ISize      &rowsAndColumns,
                       const IRectangle &initial,
                       const Style      &style )
  : IControl()
  {
  this -> initialize( rowsAndColumns,
                      id,
                      parent,
                      owner,
                      initial,
                      style );
  }
```

Attaching Handlers

Once your constructor has invoked the `startHandlingEventsFor` member function of `IWindow`, you can attach handlers to your custom control. Our value set example does not add any handlers automatically. Sometimes you may have to attach handlers to implement your custom control. Any custom control implemented strictly using C++, with no custom Presentation Manager window class, would have to attach a handler to customize the standard window you built it upon. For example:

- An `IGraphicPushButton` attaches handlers to the base `IPushButton` to manage the drawing of the button graphics.
- An `ICanvas` attaches a handler to its underlying `WC_STATIC` Presentation Manager window to manage the layout of child windows.

We cover custom handlers later in this chapter. That discussion will guide you in designing and implementing the handlers that you may need to implement your custom control. You would attach the handler in your constructors immediately after the call to `startHandlingEventsFor`.

Custom Control Styles

Most controls provide for a variety of styles of appearance and behavior. When you are designing your custom controls, you should enable programmers using your class to specify the specific style of control at construction time. You should also provide functions to permit them to change the control style after creation.

When you are encapsulating a Presentation Manager window class, you can usually determine all the control styles by examining the values documented as valid input, via the style argument on the `WinCreateWindow` call. For example, with `WC_VALUESET`, we see there are ten `VS_*` style constants defined. These include `VS_BITMAP`, `VS_ICON`, `VS_BORDER`, `VS_RIGHTTOLEFT`, and so forth.

You should define a nested `Style` bit mask class, and a set of static members corresponding to each supported style attribute of your custom control class. You generate the nested `Style` class declaration using the `IBitFlag` class and the macros defined in `IBITFLAG.HPP`. See Chapter 26, “Data Types,” for a detailed description of the use of these macros.

In our `ValueSet` example, we generate the declaration of our nested `Style` class using the macro invocation:

```
INESTEDBITFLAGCLASSDEF2( Style, ValueSet, IWindow, IControl);
```

In this example, we also declare that we wish to mix our styles with those defined for our base classes `IWindow` and `IControl`. This is so you can also use the `IWindow::visible` and `IControl::tabStop` styles with `ValueSet` objects.

We define the set of individual style attributes of a `ValueSet` as static data members of type `ValueSet::Style`.

```
static const Style
  border,
  itemBorder,
  rightToLeft,
  scaleBitmaps,
  drawItem,
  classDefaultStyle;
```

We do not bother defining styles corresponding to `VS_BITMAP`, `VS_ICON`, and so on. The reason is that these styles are really irrelevant when creating the value set. The individual value set items can be of any type, regardless of the value set style, so we simply handle this attribute in our `ValueSet::Item` class (which we describe below).

There is also a style value named `classDefaultStyle`. This is the style that describes the default characteristics of a `ValueSet` the constructors will use, unless you provide an explicit style when you create the value set object. `ValueSet`'s default style is 0; none of the style attributes is set. Here is the definition of these static members.

```
const ValueSet::Style
  ValueSet::border            = VS_BORDER,
  ValueSet::itemBorder        = VS_ITEMBORDER,
  ValueSet::rightToLeft       = VS_RIGHTTOLEFT,
  ValueSet::scaleBitmaps      = VS_SCALEBITMAPS,
  ValueSet::drawItem          = VS_OWNERDRAW,
  ValueSet::classDefaultStyle = WS_VISIBLE;
```

In this example, we initialize our style values to the corresponding constant from the Developer's Toolkit for OS/2. This simplifies the implementation of `ValueSet`, because we don't have to convert `ValueSet::Style` values to the equivalent `unsigned long` value when passing them to the Presentation Manager. For example, we pass the argument style passed to the `ValueSet` constructors straight through as an `unsigned long` to `IWindow::create`.

The User Interface Library provides flexibility regarding default styles. To be consistent with the rest of the library, you should provide static functions in your custom control class to get and set the user's choice of what the default style should really be. These functions usually look like the following.

```
static Style
  defaultStyle();
static void
  setDefaultStyle( const Style &style );
```

To implement this default style support, you simply need to do the following.

- Define a private static data member to hold the user's choice for default style.

  ```
  private:
  static Style
     dfltStyle;
  ```

- Initialize this to the value defined by the `classDefaultStyle` data member.

  ```
  ValueSet::Style
    ValueSet::dfltStyle = WS_VISIBLE;
  ```

- Implement the static functions to get and set this static data member.

  ```
  ValueSet::Style ValueSet :: defaultStyle ( )
  {
    return ValueSet::dfltStyle;
  }
  ```

```
void ValueSet :: setDefaultStyle ( const Style &style )
{
  ValueSet::dfltStyle = style;
}
```

Typically, your custom control class will need to provide a member function to test each style attribute, and two member functions to turn a given attribute on and off. For example, the `ValueSet` class provides these functions to manage the `VS_BORDER` style.

```
Boolean
  hasBorder ( ) const;

ValueSet
 &enableBorder  ( Boolean flag = true ),
 &disableBorder ( );
```

The User Interface Library convention is to provide two style-setting functions for each attribute, one to turn the attribute on—but accepting an optional `Boolean` argument to turn it off—and one to turn it off. This permits you to replace:

```
if ( needsBorder )
  aValueSet.enableBorder();
else
  aValueSet.disableBorder();
```

with a simpler version:

```
aValueSet.enableBorder( needsBorder );
```

Below is the implementation of the three `ValueSet::border` style manipulation functions.

```
Boolean ValueSet :: hasBorder ( ) const
  {
  return this->style() & VS_BORDER;
  }

ValueSet &ValueSet :: enableBorder ( Boolean flag )
  {
  if ( flag )
    this->setStyle( this->style() | VS_BORDER );
  else
    this->setStyle( this->style() & ~VS_BORDER );
  return *this;
  }

ValueSet &ValueSet :: disableBorder ( )
  {
  return this->enableBorder( false );
  }
```

Standard Member Functions

It happens that most of the virtual functions defined for `IControl` and `IWindow` objects can be applied to objects of any derived class without any difficulty. As a result, you usually do not need to override any `IWindow` or `IControl` virtual functions in your custom control classes.

The User Interface Library design does impose a small number of simple requirements your custom control class must meet to be fully integrated into the library. As the designer of a custom control class, you should make sure you address each of these issues and provide the appropriate level of support in your custom control.

Color Support

The User Interface Library provides full support for controlling the colors that the Presentation Manager uses to draw controls. You should provide the same level of support in your custom control classes.

The design strategy for manipulating window colors is to define the set of color areas, using a `ColorArea` enumeration that the control supports, and define the member functions to query, and set, the colors to be used to draw these areas. The color areas are the unique visual elements of the control. For the value set controls, the `WC_VALUESET` window class dictates these color areas found in Table 30-1.

Table 30-1. Value Set Color Areas

Color Area	Description
`borders`	This is the color used to paint the value set borders. It corresponds to the presentation parameter with code `PP_BORDERCOLOR`.
`foreground`	This is the color used to paint the foreground for text items. It corresponds to the presentation parameter with code `PP_FOREGROUNDCOLOR`.
`background`	This is the color used to paint the value set background for text, icon, and empty items. It corresponds to the presentation parameter with code `PP_BACKGROUNDCOLOR`.
`highlightBackground`	This is the color used to paint the selection emphasis around the selected value set item. It corresponds to the presentation parameter with code `PP_HILITEBACKGROUNDCOLOR`.

The declaration of the `ValueSet::ColorArea` enumeration is as follows.

```
enum ColorArea
  {
  borders,
  foreground,
  background,
  hightlightBackground
  };
```

Once you have defined your control's color areas, you can add the member functions to query and set the colors to be used to draw these areas. These functions are essentially the same in each control class; they simply use the `ColorArea` enumeration appropriate to the class. Here are the declarations of these functions for `ValueSet`.

```
virtual IColor
  color ( ColorArea area ) const;

virtual ValueSet
 &setColor ( ColorArea area,
             const IColor &color );
```

You implement these functions by translating the enumeration values to the appropriate PP_* index, and then calling the base IWindow function that accepts these indices as argument. In this example, there are of course PP_* values corresponding to each color area. If your custom control has unique color areas, then you would reuse a PP_* index that is not otherwise applicable to your control.

```
IColor ValueSet :: color ( ColorArea area ) const
  {
  unsigned long
    colorArea;
  IGUIColor
    defaultColor( IGUIColor::windowBgnd );
  switch ( area )
    {
    case borders:
      colorArea    = PP_BORDERCOLOR;
      defaultColor = IGUIColor::frameBorder;
      break;
    case foreground:
      colorArea    = PP_FOREGROUNDCOLOR;
      defaultColor = IGUIColor::windowText;
      break;
    case background:
      colorArea    = PP_BACKGROUNDCOLOR;
      defaultColor = IGUIColor::windowBgnd;
      break;
    case highlightBackground:
      colorArea    = PP_HILITEBACKGROUNDCOLOR;
      defaultColor = IGUIColor::hiliteBgnd;
      break;
    }
  return IWindow::color( colorArea, defaultColor );
  }

ValueSet &ValueSet :: setColor ( ColorArea     area,
                                 const IColor &color )
  {
  unsigned long
    colorArea;
  switch ( area )
    {
    case borders:
      colorArea    = PP_BORDERCOLOR;
      break;
    case foreground:
      colorArea    = PP_FOREGROUNDCOLOR;
      break;
    case background:
      colorArea    = PP_BACKGROUNDCOLOR;
      break;
    case highlightBackground:
      colorArea    = PP_HILITEBACKGROUNDCOLOR;
      break;
    }
  IWindow::setColor( colorArea, color );
  return *this;
  }
```

Calculating Minimum Size

The User Interface Library canvas classes' layout strategies require information about the minimum size of their child controls. If you want these canvasses to work effectively with your custom controls, then you must override the calcMinimumSize virtual member function in your custom control class. This function returns an `ISize` object that describes the minimize size at which your control can still manage to effectively display its contents.

Fortunately, your `calcMinimumSize` function does not have to provide a precise answer. A reasonable estimate, erring on the conservative side, is generally good enough. You can get a feel for how to implement this function, for your custom controls, by looking at the implementation of `ValueSet::calcMinimumSize`.

```
ISize ValueSet :: calcMinimumSize ( ) const
  {
  ISize
    dim = dimensions();
  // Find widest and tallest items...
  unsigned
    widest  = 0,
    tallest = 0;
  for ( unsigned row = 0; row < dim.height(); row++ )
    for ( unsigned col = 0; col < dim.width(); col++ )
      {
      Item
        cell = item( row+1, col+1 );
      ISize
        size;
      switch ( cell.itemType )
        {
        case ValueSet::Item::emptyItem:
          // Empty items must be at least 2x2.
          size = ISize( 2, 2 );
          break;
        case ValueSet::Item::colorItem:
          // We'll make color items at least 5x5.
          size = ISize( 5, 5 );
          break;
        case ValueSet::Item::bitmapItem:
          // We'll make bitmap items at least 50x50.
          size = ISize( 50, 50 );
          break;
        case ValueSet::Item::iconItem:
          // We'll make icon items at least 32x32.
          size = ISize( 32, 32 );
          break;
        case ValueSet::Item::textItem:
          // We'll make text items avg char size * text length.
          size = ((ValueSet*)this)->characterSize()
                   .scaledBy( cell.text().length(), 1 );
          break;
        }
      // Update maximums if this item is bigger...
      if ( size.width() > widest )
        widest = size.width();
      if ( size.height() > tallest )
        tallest = size.height();
      }
  // Calculate size that will hold all items + 12 for border.
  ISize
    min = ISize( widest  * dim.width() + 12,
                 tallest * dim.height() + 12 );
  ISize
    space = itemSpacing();
  // Add horizontal space, including 7 for border/highlighting.
  if ( dim.width() > 1 )
    min.setWidth( min.width()
                  +
                  (dim.width()-1)*( space.width() + 7 ) );
```

```
// Add vertical space, including 7 for border/highlighting.
if ( dim.height() > 1 )
  min.setHeight( min.height()
                 +
                 (dim.height()-1)*( space.height() + 7 ) );
return min;
}
```

Since all items in the value set must be the same size, we make the item size big enough for both the widest and tallest items. We then calculate the minimum size for the entire value from that item size. We also allow for the spacing between items and some room for the border around the outside of the value set.

There is plenty of room for subjectivity in this minimum-size calculation. We let bitmap items be sized to (50,50) which is much smaller than most bitmaps. We also presume all icons can fit within an item sized at (32,32). This function calculates the size of text items using the size of average characters rather than the actual characters in the item text. However, the margin of error, resulting from all these approximations, is negligible. You should only invest as little time and energy into the design of your `calcMinimumSize` functions as is necessary to get the precision required for your application's needs.

If the size calculated by this function is wrong, then you can always call `setMinimumSize` to set the minimum size of the control to some specific value. `calcMinimumSize` will not be called if you have set an explicit size by calling `setMinimumSize`. See Chapter 15, "Canvases," for more information about canvas layout.

Notifying Canvases When Minimum Size Changes

Supplying an appropriate `calcMinimumSize` function, so a canvas can properly lay out its children, if one of them is one of your custom controls, is only half the battle. You need to notify the canvas to update the layout of its children, when the minimum size of your custom control has changed. You do so by calling the `setLayoutDistored` function with the `minimumSizeChanged` flag on. Here is one example from the implementation of the sample `ValueSet` control.

```
ValueSet &ValueSet :: setDimensions ( const ISize &dimensions )
  {
  IString
    ctlData = ValueSet::controlData( dimensions );
  WNDPARAMS
    parms = { WPM_CTLDATA,
              0,
              0,
              0,
              0,
              0,
              (char*)ctlData };
  if ( sendEvent( WM_SETWINDOWPARAMS, &parms ) == 0 )
    ITHROWGUIERROR( "WM_SETWINDOWPARAMS" );
  else
    setLayoutDistorted( minimumSizeChanged, 0 );
  return *this;
  }
```

Custom Control Functions

In addition to the standard functions all controls must provide, your custom controls will likely have another set of member functions that provide services unique to your control. We will not try to tell you how to design custom controls. Instead, we presume you have designed your custom control so it is implemented via a Presentation Manager window procedure. We will describe how to encapsulate the functions supported by those windows in your custom control C++ class.

The value set control provides a good example of how to accomplish this task. To determine what member functions our `ValueSet` class should provide, we simply look at the set of Presentation Manager events a `WC_VALUESET` window supports. After just a little bit of analysis, we can come up with the following functions, which we group into three categories found in Table 30-2. For each of these events, we define one or more member functions of our `ValueSet` class to provide you access to the service from C++.

The design and implementation of the functions in the first two categories is straightforward. The arguments for each function correspond to the event parameters on the associated Presentation Manager window event. The return value from the member function matches the event result. The following example shows this mapping. It gives the implementation of the `dimensions` and `setDimensions` member functions of `ValueSet`.

Table 30-2. Value Set Functions

Message	Member Functions
Value Set Dimension	
WM_QUERYWINDOWPARAMS	dimensions numberOfRows numberOfColumns
WM_SETWINDOWPARAMS	setDimensions
VM_QUERYMETRICS	itemSize itemSpacing
VM_SETMETRICS	setItemSize setItemSpacing
Value Set Selection	
VM_QUERYSELECTED	hasSelection selection selectedItem selectedRow selectedColumn
VM_SELECTITEM	setSelection
Value Set Item Manipulation	
VM_QUERYITEM	item
VM_SETITEM	setItem setItemContents
VM_QUERYITEMATTR	item
VM_SETITEMATTR	setItem setItemContents setItemStyle

```
ISize ValueSet :: dimensions ( ) const
  {
  ISize
    result;
  IString
    ctlData = ValueSet::controlData( ISize( 0, 0 ) );
  WNDPARAMS
    parms = { WPM_CTLDATA,
              0,
              0,
              0,
              0,
              ((VSCDATA*)(char*)ctlData)->cbSize,
              (char*)ctlData };
  if ( sendEvent( WM_QUERYWINDOWPARAMS, &parms ) != 0 )
    {
    VSCDATA
     *p = (VSCDATA*)( parms.pCtlData );
    result = ISize( p->usColumnCount, p->usRowCount );
    }
  else
    ITHROWGUIERROR( "WM_QUERYWINDOWPARAMS" );
  return result;
  }

ValueSet &ValueSet :: setDimensions ( const ISize &dimensions )
  {
  IString
    ctlData = ValueSet::controlData( dimensions );
  WNDPARAMS
    parms = { WPM_CTLDATA,
              0,
              0,
              0,
              0,
              0,
              (char*)ctlData };
  if ( sendEvent( WM_SETWINDOWPARAMS, &parms ) == 0 )
    ITHROWGUIERROR( "WM_SETWINDOWPARAMS" );
  return *this;
  }
```

In addition to the basic support for the various Presentation Manager events, you should apply conventional C++ class library design techniques to make programming your custom controls easier. As an example, we have added an additional `hasSelection` member function to our `ValueSet` control. This function returns a `Boolean` indicator of whether the value set has a currently selected item. Users can call this function instead of using the basic `selection` function, and testing its return value to see if it equals `IPoint(0,0)`. You should also use overloading and default arguments, where applicable, to simplify use of your custom control class.

`ValueSet`'s item query and set functions are a little more complicated. One objective of encapsulating windows as C++ objects is to raise the level of abstraction. This requires us to combine sets of related data to form objects, and, when necessary, we give these objects functionality you can apply to the data. For our `ValueSet` control, we combine the separate item content and item attribute characteristics, which the Presentation Manager control supports, into the nested class `ValueSet::Item`.

As a consequence of the additional abstraction, provided by the `ValueSet::Item` class, the `ValueSet` member functions that manipulate items do not match the Presentation Manager events quite as closely as do the simpler `ValueSet` functions. For example, the `setItem` member function will set both the item contents and the item style. The various `setContents` member functions take care of resetting the item attribute. The attribute must first be set to indicate the type of contents we will assign.

Below is the implementation of the `ValueSet` functions, which set the contents of an item to a bitmap loaded from a resource file. You should be able to see the value provided by the C++ wrapper as compared to programming the equivalent operation in plain Presentation Manager functions:

```
ValueSet
 &ValueSet :: setItemContents ( unsigned long      row,
                                unsigned long      column,
                                const IResourceId &resId,
                                ItemType           type )
  {
  const IResourceLibrary
   &resLib = resId.resourceLibrary();
  switch ( type )
    {
    case bitmapItem:
      setItemContents( row,
                       column,
                       resLib.loadBitmap( resId.id() ) );
      break;
    case iconItem:
      setItemContents( row,
                       column,
                       resLib.loadIcon( resId.id() ) );
      break;
    case textItem:
      setItemContents( row,
                       column,
                       resLib.loadString( resId.id() ) );
      break;
    }
  return *this;
  }

ValueSet
 &ValueSet :: setItemContents ( unsigned long row,
                                unsigned long column,
                                const IBitmapHandle &bmp )
  {
  setItemAttributes( row,
                     column,
                     IEventParameter2( VIA_BITMAP, true ) );
  setItem( row,
           column,
           IEventParameter2( bmp ) );
  return *this;
  }

void ValueSet :: setItemAttributes
                   ( unsigned long           row,
                     unsigned long           column,
                     const IEventParameter2 &attr )
  {
  IEventParameter1
    coord( row, column );
  if ( sendEvent( VM_SETITEMATTR,
                  coord,
                  attr ) == 0 )
    ITHROWGUIERROR( "VM_SETITEMATTR" );
  }
```

```
void ValueSet :: setItem ( unsigned long           row,
                           unsigned long           column,
                           const IEventParameter2 &contents )
  {
  IEventParameter1
    coord( row, column );
  if ( sendEvent( VM_SETITEM,
                  coord,
                  contents ) == 0 )
    ITHROWGUIERROR( "VM_SETITEM" );
  }
```

All this implementation code is necessary to enable you to use `ValueSet` controls more conveniently. It permits you to set a bitmap using the simple statement:

```
valueSet.setItemContents( BITMAP_ID, ValueSet::bitmapItem );
```

As a user, you would not have to know how the value set accomplishes this simple request.

Custom Handlers

Custom controls will typically send a set of control-specific notification events in response to user input such as mouse clicks, key presses, and so on. This section will describe how you design and implement custom handler and event classes to encapsulate those notification events and the processing of them. Programmers can then derive from these custom handler classes, override virtual functions, and process the custom control events to tailor use of the customer controls for their applications.

The first step in designing the custom handlers and events for your custom control windows, is to analyze the set of owner notification messages that the control broadcasts. After this analysis, you should be able to place each event into one of two categories.

- Notification events that are identical, or at least very similar, to similar events, which occur for other controls.
- Notification events specific to your custom control.

Events falling into the first category should be dealt with by extending the existing handler and event classes, which already encapsulate those events for the existing controls. The second set will require you to derive a new class from `IHandler` and also derive one, or more, new class(es) from `IEvent`.

Let us look at the value set control messages and perform the preliminary analysis of its notification events. We will arrive at a set of handler and event classes that will permit you to tailor the handling of these events. There are ten separate notification events that a value set issues. However, half of these deal with direct manipulation. We will cover those in Chapter 21, "Direct Manipulation." Table 30-3 lists five events, based on the `notifyCode` parameter of the `WM_CONTROL` event, which apply to value set controls.

It happens that we do not need special handler or event classes for any of these events, except for the `VN_HELP` notification event. Even though the User Interface Library does not provide its own value set control class, its `ISelectHandler` and `IFocusHandler` objects properly detect and process the select, enter, and focus change notification events. If you attach one of these handlers to a value set control window, you can process these events.

Table 30-3. Value Set Notification Events

Notify Code	Description
VN_ENTER	The user has pressed the Enter key while the value set had the input focus.
VN_SELECT	The user has selected one of the value set items.
VN_KILLFOCUS	The user has switched the input focus from the value set to another window.
VN_SETFOCUS	The user has switched the input focus to the value set from another window.
VN_HELP	The user has pressed the Help key (F1) while the value set had the input focus.

We do need to design a handler to enable you to process help notification events. Writing a custom handler to process some set of events involves the following steps.

- Deriving from the appropriate `IHandler` base class. Usually, you derive straight from `IHandler`. In this example, we will create a `ValueSetHandler` class with `IHandler` as a public base class.
- Add an overloaded version of `startHandlingEventsFor` that accepts as argument a reference to the type of control to which your handler can be attached. In addition, you should override the generic version of the function, which attaches the handler to a generic `IWindow`, to disable it. We make this function private in our `ValueSetHandler` class, to make it even harder for somebody to attach our handler to something other than a `ValueSet` control.
- For convenience, we also add a constructor that takes a `ValueSet` as argument. Since the most common use of a `ValueSetHandler` is for you to create one, and immediately attach it to a `ValueSet`, this constructor makes it possible to do both of those things in one step.
- Provide virtual functions for each event to be handled. When programmers wish to implement their application-specific event handling, they simply override these functions.
- Most importantly, you must override the `dispatchHandlerEvent` function to detect when the events of interest occur. The implementation of this function will look a lot like a conventional Presentation Manager window procedure. You switch on the event identifier, and the notification code, if it is a `WM_CONTROL` owner notification event. In each case statement, you create a more specific type of event and dispatch the appropriate virtual function. Custom events will be discussed in more detail later.

Even though you can handle most of the value set notification events using standard User Interface Library handlers, we still add support for the complete set of value set notification events to `ValueSetHandler`. There are two reasons for this.

First, programmers will generally produce two kinds of handlers. They will build general purpose ones, which work for many different kinds of controls. The User Interface Library direct manipulation handlers are examples of these. They also will build custom controls using just C++ to provide all the tailoring in a single handler. To facilitate development of specialized value sets, we let you handle all the value set notifications in one place.

Secondly, there are often subtle differences between notification events, depending on their source. For example, the value set's "enter" and "select" events both provide the row and column of the selected item in the second event parameter. The only way you can obtain these from the generic `IControlEvent` object in the generic `enter`, or `selected` handler functions, is to extract this parameter as an `unsigned long` and extract the values from it. We can do better in a `ValueSetEvent` class by adding explicit row and column member functions. To put these events to any use, requires us to create `ValueSetEvent`s and provide overloaded versions of `enter` and `selected`, which get a `ValueSetEvent` as argument.

So, our `ValueSetHandler` class has five virtual functions corresponding to the five (non-drag/drop) value set owner notification events.

Custom Events

Now we will discuss those events themselves in a little more detail. All event classes are, essentially, wrappers for the five pieces of data that make up a User Interface Library event. They are:

- A source window handle. This data can be translated to a generic `IWindow` pointer.
- An event identifier.
- Two message parameters; the value and meaning of which depend on the specific event. These simply are `unsigned long`s (or `void *` pointers) by default.
- The event result to be passed back to the sender of the event.

Since these comprise the state data of all event objects, a generic `IEvent` can be used to represent any event. Generally, you will want to create derived `IEvent` classes, whenever you need to make it easier to translate the generic event attributes into something more meaningful. For our value set custom control, we create an enhanced `ValueSetEvent` class that:

- Returns the control window pointer as type `ValueSet*`.
- Translates the second event parameter as row and column number.

The code listing at the end of this chapter has the full declaration of our `ValueSetHandler` and `ValueSetEvent` classes.

Event Results

While it does not come up in the implementation of the sample `ValueSet` custom control, there are some additional issues you should be aware when dealing with the result value associated with the handling of certain events.

Some Presentation Manager events require the handler of the event return a result. In terms of the User Interface Library, this means the handler that processes the event must set the result attribute of the `IEvent` object, which gets passed to the handler's `dispatchHandlerEvent` function. This imposes two requirements on your custom handler and event classes.

First, you should provide specialized functions to set the event attribute of your custom event classes. While a programmer can always call the generic `setResult` function declared in `IEvent`, its function doesn't provide any clues as to whether or not they must call it, or indicate what value the result should be set to. As an example, if we were to define a `ValueSetDrawEvent` class, to represent owner draw events for value sets, we would have the class provide functions like:

```
Boolean
  hasItemBeenDrawn ( ) const;
ValueSetDrawEvent
 &itemDrawn ( Boolean drawn = true );
```

Second, as we saw in this chapter, you handle most events in an overridden virtual function, which the handler's `dispatchHandlerEvent` calls. The usual scenario has the form:

```
...
case WM_DRAWITEM:
  {
  // Construct specialized event from generic one and
  // dispatch handler function:
  ValueSetDrawEvent
    customEvent( event );
  this -> draw( customEvent );
  break;
  }
...
```

In this example, `ValueSetDrawEvent` is the custom event class and `draw` is the handler virtual function that processes the event.

So what if we override `draw` and draw the item? Somehow, we need this fact to get back to the other handlers in the chain, and back to the Presentation Manager value set itself, so they know not to draw it. The first step is to add the `itemDrawn` function to the event—as we described above—and to call that function when you draw the item. However, more needs to be done. The above code cannot work, as it discards any result you might have placed in the `ValueSetDrawEvent`.

The solution is to transfer the event result back to the generic `IEvent` object, which the User Interface Library passed to `dispatchHandlerEvent`. This requires a single line of code to assign to the generic event result, after processing the specialized event.

```
...
this -> draw( customEvent );
event.setResult( customEvent.result );
...
```

Summary

This chapter has described the process you use to incorporate custom controls into the User Interface Library. The basic steps are:

- To create a new class somewhere in the window class hierarchy, usually derived directly from `IControl`. Your custom control class should provide:
  - Constructors that permit new custom control windows to be created, and existing windows to be encapsulated as C++ objects.
  - A nested `Style` class to describe the style attributes of your custom control windows, and a set of member functions to allow a programmer to query and set the various attributes.
  - A `ColorArea` enumeration, which defines the various drawing components of your control, and member functions `color` and `setColor` to get and set the colors used to draw these components.
  - An override of the `calcMinimumSize` function, which calculates the minimum size your control needs to display its contents.
  - Member functions corresponding to the set of additional Presentation Manager window events to which your custom controls respond.
- A custom handler class derived from `IHandler`, which handles the owner notification events your control generates. The handler will invoke separate virtual handler functions for each event.
- One or more custom event classes to encapsulate any special event parameters, or results, which your control's notification events require.

The complete definition of the classes that implement the value set control, and a test program, are contained on the example program disk. Table 30-4 indicates the location of the value set components, after you install these programs.

Table 30-4. Value Set Components

Component	Example Location
ValueSet Interface	extlib\valueset\valueset.hpp
ValueSet Implementation	extlib\valueset\valueset.cpp
ValueSetHandler Interface	extlib\valueset\vsethdr.hpp
ValueSetHandler Implementation	extlib\valueset\vsethdr.cpp
ValueSetEvent Interface	extlib\valueset\vsetevt.hpp
ValueSetEvent Implementation	extlib\valueset\vsetevt.cpp
ValueSet Test Program	extlib\valueset\testvset.cpp