Learning to Program with Haiku

Lesson 18

Written by DarkWyrm

All material © 2010 DarkWyrm

With buttons and menus out of the way, we will start learning more about some of the other kinds of window controls available to us, such as the following controls:

Control	Description
BAlert	A pop-up message. They're great for the occasional error message, but they are also very easy to overuse and abuse. Use them sparingly.
BBox	A BView which draws a box around other child controls to visually group related controls together. It also can optionally have a text label in the top left corner.
BButton	A button with a text label which sends a message when clicked.
BChannelSlider	A slider class which supports multiple channels. It's mostly used with the Media Kit.
BCheckBox	A standard-issue checkbox which can send a message when clicked.
BColorControl	A customizable color picker. The main drawback is that it takes up quite a lot of space.
BListView	A list of items. The items themselves are sufficiently customizable to show just about anything but are often BStringItems to display a list of names.
BMenu	A container for menu items.
BMenuBar	A container for menus intended to be placed at the top of a window.
BMenuField	A button-like menu container which shows a pop-up menu. Getting them to resize properly can be problematic.
BMenuItem	A clickable item in a menu.
BOptionPopUp	A convenience class which quickly sets up a BMenuField.
BOutlineListView	A hierarchical list, but otherwise no different from BListView.
BPictureButton	A button which uses a BPicture vector graphic to draw a picture for a label. They can be one-state buttons like BButton or two-state buttons that stay down when clicked and pop back up when clicked again.
BPopUpMenu	A special type of menu that can be shown anywhere on the screen.
BRadioButton	The circular "checkbox" that enables the user to choose one from several options.
BScrollBar	A single scrollbar used to scroll BViews.
BScrollView	A convenience class for attaching scrollbars to BViews and associated controls.
BSeparatorItem	A menu item which displays a horizontal line. They are used to separate groups of menu items.
BSlider	A highly-customizable horizontal slider.
BStatusBar	A progress bar class.
BStringView	A static text label.

Control	Description
BTabView	Group together sheets of controls on tabs. It requires a little more care to use them than what it might initially seem.
BTextControl	A single line text editing control.
BTextView	A multiline text editing control with limited word processing capabilities.
BView	The base class from which all controls are derived. It has a lot of methods and it can be a little overwhelming, but it's powerful and flexible.

That's a lot of different classes! At first, it seems pretty overwhelming, but it's not too bad after a little tinkering, especially when you consider that there is quite a bit they have in common. You might want to keep this control table handy for your first projects. We're not going to look at every control in detail, but the ones that we skip should be easy enough to pick up on your own.

Typecasting Revisited

Way back in lesson 7 we learned about typecasting, i.e. reinterpreting the data in a variable as a different type. One of the problems with casting in C is that it isn't safe – whenever you cast a variable, it always succeeds. If you choose the wrong class to cast, it still succeeds, typically with unpredictable results. C++ offers different kinds of casting. The C++ way should be preferred unless you're writing code in C. Here they are:

```
static_cast<TypeToCastTo *>(pointerToCast);
```

Static casts are used to convert one type to another. It relies upon compile-time information and is often used for the purposes that the regular C casting method is used – changing pointer types and arithmetic conversions. It works so long as the language supports the conversion between the two types.

```
dynamic_cast<TypeToCastTo *>(pointerToCast);
```

Aside from static casts, dynamic casts will be one of your most commonly-used cast types when working with Haiku. They are used to safely navigate an inheritance hierarchy. If no inheritance links together the "from" type and the "to" type, `NULL` will be returned instead. Thus, you can cast to only those pointer types you're supposed to be able cast. We will use one of these in today's project.

```
const_cast<TypeToCastTo *>(pointerToCast);
```

Const casting adds or removes the changeability of a pointer. For example, there are instances where a function will take a non-`const` parameter that it doesn't change. Passing a constant pointer to one of these functions will require a const cast.

```
reinterpret_cast<TypeToCastTo *>(pointerToCast);
```

A reinterpret cast is used only rarely. It converts a pointer from one type to another, regardless of whether the two pointers are related or not. Almost all of the tasks which a reinterpret cast can do can be done with a static cast, and the remaining conversions are almost always not portable, so this kind of cast should be used only when absolutely necessary, such as converting between function pointer types.

Project: Using List Controls

This project, called ListTitle, will be about using list controls, namely the BListView class. One paradigm found in the Haiku API is that list-based controls use lightweight items. BMenu and BListView actually do most of the work and BMenuItem and BListItem are actually very simple classes. Designing these classes this way saves memory.

1. Create a new project in Paladin, but this time use the "GUI with Main Window" template to save some typing – this template creates the boilerplate code for the App and MainWindow classes.
2. Our project and target name will be ListTitle.
3. Once created, open App.cpp, change the MIME signature of the app to "application/x-vnd.test-ListTitle", and close it.
4. Open MainWindow.h, add the include ListView.h to the top of the header.
5. Add a `private:` access section keyword at the bottom of the MainWindow class definition.
6. In the private section of the MainWindow definition, declare the property `BListView *fListView`. We will be using fListView in more than just the window constructor, so it makes sense to stash away a pointer to it for later use.

Now let's get down to business: setting up the MainWindow's controls and making them do something. Open up MainWindow.cpp and change it to this code:

```
#include "MainWindow.h"

#include <Button.h>
#include <ListItem.h>
#include <ScrollView.h>

enum
{
	M_RESET_WINDOW = 'rswn',
	M_SET_TITLE = 'sttl'
};

MainWindow::MainWindow(void)
	:	BWindow(BRect(100,100,500,400),"The Weird World of Sports",
			B_TITLED_WINDOW, B_ASYNCHRONOUS_CONTROLS |
					B_QUIT_ON_WINDOW_CLOSE)
{
	// Here we will make a BView that covers the white area inside the window so
	// we can choose a background color. You'll want to do this in the windows
	// of your projects -- your projects will look more professional
	BRect r(Bounds());
	BView *top = new BView(r,"topview",B_FOLLOW_ALL,B_WILL_DRAW);
	AddChild(top);

	// ui_color() returns a system color, such as the window tab color, menu
	// text color, and so forth. The Panel Background color is the one used
	// for background views like this one.
	top->SetViewColor(ui_color(B_PANEL_BACKGROUND_COLOR));

	// Create a button and place it at the bottom right corner of the window.
	// The BRect that we use for the BButton's frame is empty because we're
	// going to have it resize itself and then move it to the corner based on
```

```
// the actual size of the button, so it's pointless to specify a size
BButton *reset = new BButton(BRect(),"resetbutton","Reset",
                             new BMessage(M_RESET_WINDOW),
                             B_FOLLOW_RIGHT | B_FOLLOW_BOTTOM);
top->AddChild(reset);
reset->ResizeToPreferred();

// Place the button in the bottom right corner of the window with 10 pixels
// of padding between the button and the window edge. 10 pixels is kind of a
// de-facto standard for control padding. It's enough that controls don't
// look crowded without taking up tons of space.
reset->MoveTo(Bounds().right - reset->Bounds().Width() - 10.0,
                        Bounds().bottom - reset->Bounds().Height() - 10.0);

r = Bounds();
r.InsetBy(10.0,10.0);

// When working with BScrollViews, you must compensate for the width/height
// of the scrollbars when determining the size of the control that we will
// attach to the BScrollView. B_V_SCROLL_BAR_WIDTH is a defined constant for
// the width of the vertical scroll bar.
r.right -= B_V_SCROLL_BAR_WIDTH;

// Frame works like Bounds() except that it returns the size and location of
// the control in the coordinate space of the parent view. This will make
// fListView's bottom stop 10 pixels above the button.
r.bottom = reset->Frame().top - 10.0 - B_H_SCROLL_BAR_HEIGHT;

// Most of these parameters are exactly the same as for BView except that we
// can also specify whether the user is able to select just 1 item in the
// list or multiple items by clicking on items while holding a modifier key
// on the keyboard.
fListView = new BListView(r,"colorlist",B_SINGLE_SELECTION_LIST,
                        B_FOLLOW_ALL);

// We didn't call AddChild on fListView because our BScrollView will do that
// for us. When created, it creates scrollbars and targets the specified
// view for any scrolling they do. When the BScrollView is attached to the
// window, it calls AddChild on fListView for us.

// If we call AddChild on fListView before we create this scrollview, our
// program will drop to the debugger when we call AddChild on the
// BScrollView -- a BView can have only one parent.
BScrollView *scrollView = new BScrollView("scrollview",fListView,
                                    B_FOLLOW_ALL,0,true,true);
top->AddChild(scrollView);

// A BListView's selection message is sent to the window any time that the
// list's selection changes.
fListView->SetSelectionMessage(new BMessage(M_SET_TITLE));

fListView->AddItem(new BStringItem("Toe Wrestling"));
fListView->AddItem(new BStringItem("Electric Toilet Racing"));
fListView->AddItem(new BStringItem("Bog Snorkeling"));
fListView->AddItem(new BStringItem("Chess Boxing"));
fListView->AddItem(new BStringItem("Cheese Rolling"));

fListView->AddItem(new BStringItem("Unicycle Polo"));
```

```
}


void
MainWindow::MessageReceived(BMessage *msg)
{
	switch (msg->what)
	{
		case M_RESET_WINDOW:
		{
			fListView->DeselectAll();
			break;
		}
		case M_SET_TITLE:
		{
			int32 selection = fListView->CurrentSelection();

			if (selection < 0)
			{
				// This code is here because when we press the Reset
				// button, the selection changes and an M_SET_TITLE
				// message is sent, but because nothing is selected,
				// CurrentSelection() returns -1.
				SetTitle("The Weird World of Sports");
				break;
			}

			BStringItem *item = dynamic_cast<BStringItem*>(
									fListView->ItemAt(selection));
			if (item)
				SetTitle(item->Text());
			break;
		}
		default:
		{
			BWindow::MessageReceived(msg);
			break;
		}
	}
}
```

There's not really that much to this project that's terribly different from the last one. By calling the BListView's `SetSelectionMessage()` method, we cause the title to be updated any time the user clicks on an item in the list. Most of the time we use a BListView we won't use this method. It's much more common to call a related one: `SetInvocationMessage()`, which sends a message whenever the user double-clicks on an item. Note that `DeselectAll()` also causes a selection message to be sent even though there isn't a selection, so it is necessary to handle the case when `CurrentSelection()` returns a negative value, signifying no selection.

Hopefully you're getting a feel for how BViews and regular controls are put together in applications. Most of them require a `BRect` for the size and location, a `const char *` for the name of the control, and two integers for the resizing mode and some behavior flags. Many classes also have a label and a message that is sent when the control is changed or invoked, especially those derived from BControl. Once a control is created, it is attached to the BWindow or a BView via `AddChild()`. The message sent by the control is often sent to the window to which the control is attached, but it can be redirected to

another target, such as the parent BView or to the global BApplication.

Classes and Methods to Remember

BControl

- `ResizeToPreferred(void)` – A derived class will resize itself to a good size to display its label and content.
- `SetLabel(const char *label)` / `const char * Label(void)` – Methods to get and set the label for a child class.
- `SetTarget(BHandler *handler, BLooper *looper)` – Send the invocation message to a different target, such as a BView, BWindow, or BApplication.
- `void SetEnabled(bool enabled)` / `bool IsEnabled(void)` – Methods to get and set the enabled/disabled status of a control.

BListView

- `AddItem(BListItem *item)` – Add an item to the list.
- `int32 CountItems(void)` – Returns the number of items in the list.
- `BListItem * RemoveItem(int32 index)` – Removes and returns the item at the specified index or `NULL` if there isn't one.
- `void RemoveItem(BListItem *item)` – Removes the specified item from the list. If the list doesn't have the item, it doesn't do anything.
- `int32 CurrentSelection(int32 index = -1)` – Returns the index of the currently selected item or -1 if there is no selection. The `index` argument is used to get all the selected items in a list which supports multiple item selections. A `while()` loop is generally used to get all the item indices and it normally exits when -1 is returned.
- `void Select(int32 index, bool extend = false)` – Select the item at the specified index. If `extend` is false, all other selected items are deselected before the specified item is selected.
- `void Select(int32 start, int32 end, bool extend = false)` – Selects all items from `start` to `end`. If extend is false, all other selected items are deselected before the specified items are selected.
- `void DeselectAll(void)` – deselects all items in the list.