Programming with Haiku

Lesson 19

Written by DarkWyrm

In our previous lesson we explored the Haiku scripting API and how it could be manipulated with the `hey` Terminal command. Now we will go deeper and get into the mechanics of using C++ for scripting.

More About Haiku Messaging

At first, the thought of using C++ to manipulate another program via the scripting API might seem pointless, but there are reasons to do so. One of them is for the sheer power it affords. `hey` has its limits, the most significant of which is its inability to use the Execute command and the need to escape certain kinds of data in order to get around the `bash` shell. Using C++ for scripting requires an understanding of the Haiku API's messaging classes that goes a little further than passing acquaintance, but it confers the full power made available by the scripting interface.

Most GUI development doesn't involve much mucking around with the messaging classes because the API has been designed well. The vast majority of the time we're responding to messages and explicitly sending them only once in a while. Let's take a quick peek at the messaging classes which we'll use for scripting.

BHandler and BLooper do the grunt work for message handling. BHandlers respond to messages. BLoopers receive a message and pass it through the list of attached BHandlers until one of them decides to do something with it, but because the BLooper class is a subclass of BHandler, BLoopers can also respond to a message. This happens most often when sending a message to a BWindow and its BView-based controls – BWindow is a subclass of BLooper and BView is a direct subclass of BHandler.

BMessenger is the means of transportation for all of these scripting messages getting sent around. It targets either a BLooper or BHandler, regardless of if its target is in your program or somewhere out there in the system, and provides a way to send messages to this target.

C++ Scripting

Scripting with C++ is not that different from using `hey`. More typing is needed, but the concepts are almost exactly the same: each `hey` command is a single message, the command is merely the BMessage's `what` property, and everything else is a specifier manipulated by BMessage's specifier-related methods.

Let's start by translating a `hey` command – which has a more human-friendly interface – to C++. Here it is:

```
hey StyledEdit get Title of Window 1
```

This asks StyledEdit for the title of its first document window. There are three pieces of data to use: the target, the scripting action, and the list of specifiers. Code to send this message looks like this:

```
#include <Message.h>
#include <Messenger.h>
#include <String.h>

#include <stdio.h>
```

```
int
main(void)
{
	status_t status;
	// Point the messenger at the StyledEdit application. The status
	// parameter is set to an error condition if StyledEdit is not
	// running.
	BMessenger messenger("application/x-vnd.Haiku-StyledEdit", -1,
				&status);
	if (status == B_OK)
	{
		// Set the command
		BMessage msg(B_GET_PROPERTY), reply;

		// Set the specifiers. Note that just like the hey command,
		// they are added in order of most specific to least.
		msg.AddSpecifier("Title");
		msg.AddSpecifier("Window", 1L);
		if (messenger.SendMessage(&msg, &reply) == B_OK)
		{
			// Just about any time a scripting message returns an
			// error, it is the constant B_MESSAGE_NOT_UNDERSTOOD.

			BString title;
			if (reply.FindString("result", &title) == B_OK)
				printf("Title of StyledEdit Window 1: %s\n",
					title.String());
			else
				printf("Couldn't get title of StyledEdit Window"
					" 1\n");
		}
	}
	else
		printf("StyledEdit does not appear to be running.\n");

	return 0;
}
```

As you can see, it's not any more difficult to send messages from C++ even if it involves more typing. Understanding the why and how of scripting for Haiku is the hardest part. Getting the suites from a target is involved, but not difficult. The code below obtains the handled suites for the first document window in StyledEdit, parses them, and prints them out in a shorter, more English-like format. Observe:

```
#include <Message.h>
#include <Messenger.h>
#include <PropertyInfo.h>
#include <String.h>

#include <stdio.h>

int
main(void)
{
    // C++ version of "hey StyledEdit get Suites of Window 1"
    status_t status;
    BMessenger messenger("application/x-vnd.Haiku-StyledEdit", -1,
                &status);
```

```
if (status == B_OK)
{
    BMessage msg(B_GET_PROPERTY), reply;
    msg.AddSpecifier("Suites");
    msg.AddSpecifier("Window", 1L);
    if (messenger.SendMessage(&msg, &reply) == B_OK)
    {
        // Now that we have the suites, let's parse them and
        // print them in a format that is a little easier to
        // understand than what BPropertyInfo::PrintToStream()
        // spits out.
        int32 i = 0;
        BString suiteName;
        BPropertyInfo propInfo;
        while (reply.FindString("suites", i, &suiteName) == B_OK)
        {
            printf("Suite %s:\n", suiteName.String());
            if (reply.FindFlat("messages", i, &propInfo) !=
                B_OK)
            {
                i++;
                continue;
            }

            int32 propCount = propInfo.CountProperties();
            const property_info *info = propInfo.Properties();

            for (int32 j = 0; j < propCount; j++)
            {
                BString commands, specifiers;

                int32 cmdIndex = 0;
                if (info[j].commands[0] == 0)
                    commands = "Get,Set,Count,"
                               "Create,Delete";
                else
                    while (info[j].commands[cmdIndex])
                    {
                        BString cmdLabel;
                        switch (info[j].commands[cmdIndex])
                        {
                            case B_COUNT_PROPERTIES:
                            {
                                cmdLabel = "Count";
                                break;
                            }
                            case B_CREATE_PROPERTY:
                            {
                                cmdLabel = "Create";
                                break;
                            }
                            case B_DELETE_PROPERTY:
                            {
                                cmdLabel = "Delete";
                                break;
                            }
                            case B_EXECUTE_PROPERTY:
                            {
                                cmdLabel = "Execute";
```

```
                    break;
                }
                case B_GET_PROPERTY:
                {
                    cmdLabel = "Get";
                    break;
                }
                case B_SET_PROPERTY:
                {
                    cmdLabel = "Set";
                    break;
                }
                default:
                    break;
            }

            if (cmdLabel.CountChars())
            {
                if (cmdIndex > 0 &&
                   commands.CountChars() > 0)
                    commands << ",";
                commands << cmdLabel;
            }

            cmdIndex++;
        } // end while (commands)

    if (commands.CountChars() == 0)
        commands = "None";

    int32 specIndex = 0;
    if (info[j].specifiers[0] == 0)
        specifiers = "All";
    else
        while (info[j].specifiers[specIndex])
        {
            BString label;
            switch (info[j].specifiers[specIndex])
            {
                case B_DIRECT_SPECIFIER:
                {
                    label = "Direct";
                    break;
                }
                case B_NAME_SPECIFIER:
                {
                    label = "Name";
                    break;
                }
                case B_ID_SPECIFIER:
                {
                    label = "ID";
                    break;
                }
                case B_INDEX_SPECIFIER:
                {
                    label = "Index";
                    break;
                }
```

```
                                case B_REVERSE_INDEX_SPECIFIER:
                                {
                                    label = "ReverseIndex";
                                    break;
                                }
                                case B_RANGE_SPECIFIER:
                                {
                                    label = "Range";
                                    break;
                                }
                                case B_REVERSE_RANGE_SPECIFIER:
                                {
                                    label = "ReverseRange";
                                    break;
                                }
                                default:
                                    break;
                            }

                            if (label.CountChars())
                            {
                                if (specIndex > 0 &&
                                        specifiers.CountChars() > 0)
                                    specifiers << ",";
                                specifiers << label;
                            }

                            specIndex++;
                        } // end while(specifiers)

                    if (specifiers.CountChars() == 0)
                        specifiers = "None";

                    printf("%s: %s (%s)\n", info[j].name,
                            commands.String(), specifiers.String());

                    if (info[j].usage && strlen(info[j].usage) > 0)
                        printf("\t%s\n", info[j].usage);
                } // end for each property

                printf("\n");

                i++;
            } // end for each suite
        } // end while each suite name
    } // end if status == B_OK
    else
        printf("StyledEdit does not appear to be running.\n");

    return 0;
}
```

Talk about a lot of code to do something that doesn't amount to much! This is the most complicated part of Haiku scripting. Of course, if the suite you are looking for is one of the standard ones defined by the Haiku API, then all of this work is not even necessary.

Implementing Scripting Support

Making your own classes respond to scripting messages beyond the system defaults is a great way to make your programs even more powerful and enable possibilities that you may or may not have even imagined. Depending on what you wish to do, implementing them may require more than a little work. We'll use our ColorWell control for examples of what can be done.

Two requirements must be met to implement additional scripting support: an object must be a subclass of BHandler and it must implement three key BHandler hook functions: `GetSupportedSuites()`, `MessageReceived()`, and `ResolveSpecifier()`. The exact steps taken look like this:

1. Create a static `property_info` structure to hold the description of your control's suite.
2. Implement `GetSupportedSuites()`.
3. Tweak your control's `MessageReceived()` to test for messages with specifiers and handle them separately from regular messages.
4. Write `ResolveSpecifier()`.

An array of `property_info` structures are used to define your control's scripting interface. The definition of the structure looks like this:

```
struct property_info
{
	// The name of the property this structure describes.
	char *name;

	// A zero-terminated list of supported commands. Use a zero for the
	// first command to act as a wildcard which matches any command.
	uint32 commands[10];

	// A zero-terminated list of supported specifiers. Use a zero for the
	// first specifier to act as a wildcard which matches any specifier.
	uint32 specifiers[10];

	// A string which describes the property.
	char *usage;

	// Extra space for your own use, if you like. The system won't
	// touch it.
	uint32 extra_data;
};
```

For our ColorWell class, we will define four properties: `IsRound`, a boolean value which sets the style to round if true and rectangular if false, and three integer properties: `Red`, `Green`, and `Blue`. These three will support getting and setting of the individual color values of the ColorWell's color. Here is the resulting suite definition:

```
static property_info sColorWellProperties[] =
{
	{	"IsRound", { B_GET_PROPERTY, B_SET_PROPERTY, 0 },
		{ B_DIRECT_SPECIFIER, 0 },
		"True if the color well is round, false if rectangular.", 0,
		{ B_BOOL_TYPE }
	},
	{	"Red", { B_GET_PROPERTY, B_SET_PROPERTY, 0 },
```

```
                { B_DIRECT_SPECIFIER, 0 },
                "The red value for the color well.", 0,
                { B_INT32_TYPE }
        },
        {       "Green", { B_GET_PROPERTY, B_SET_PROPERTY, 0 },
                { B_DIRECT_SPECIFIER, 0 },
                "The green value for the color well.", 0,
                { B_INT32_TYPE }
        },
        {       "Blue", { B_GET_PROPERTY, B_SET_PROPERTY, 0 },
                { B_DIRECT_SPECIFIER, 0 },
                "The blue value for the color well.", 0,
                { B_INT32_TYPE }
        },
};
```

Now that the suite definition is done, writing `GetSupportedSuites()` is a piece of cake. It will almost always do nothing more than add the name of the suite to the message it is given, add the static property list as a flattened BPropertyInfo instance, and return the parent class' version of the function.

```
BHandler *
ColorWell::ResolveSpecifier(BMessage *msg, int32 index,
                                BMessage *specifier, int32 what,
                                const char *property)
{
        BPropertyInfo propertyInfo(sColorWellProperties);
        if (propertyInfo.FindMatch(msg, index, specifier, what, property)
                        >= 0)
                return this;

        return BControl::ResolveSpecifier(msg, index, specifier, what,
                                                property);
}
```

Don't worry about the BPropertyInfo object in the sample. This class does about as little as it can get away with – it wraps around the `property_info` structure to provide some convenience functions, the most useful of which are `Flatten()`, `Unflatten()`, and `FindMatch()`.

The go-between work which bridges the scripting interface's properties to your control is done in `MessageReceived()`. Luckily, the glue code is nice and simple.

```
void
ColorWell::MessageReceived(BMessage *msg)
{
        // Our ColorWell class doesn't handle any special messages, so if
        // a message doesn't have any specifiers, we just pass it to the
        // parent class' version.
        if (!msg->HasSpecifiers())
                BControl::MessageReceived(msg);

        // Scripting depends on a reply message.
        BMessage reply(B_REPLY);

        // These variables will hold information about the current specifier.
        status_t status = B_ERROR;
```

```
int32 index;
BMessage specifier;
int32 what;
const char *property;

if (msg->GetCurrentSpecifier(&index, &specifier, &what, &property)
            != B_OK)
      return BHandler::MessageReceived(msg);

// FindMatch() searches its property_info array for a property
// which matches the specifiers in the message. It returns an index
// to the element which matches or -1 if a match was not found.
BPropertyInfo propInfo(sColorWellProperties);
switch (propInfo.FindMatch(msg, index, &specifier, what, property))
{
      // These cases are the glue code which make each property
      // do something. Just like a regular MessageReceived() case,
      // it passes unrecognized properties to the parent class.
      case 0:     // IsRound
      {
            if (msg->what == B_SET_PROPERTY)
            {
                  bool isRound;
                  if (msg->FindBool("data", &isRound) == B_OK)
                  {
                        SetStyle(isRound ? COLORWELL_ROUND_WELL :
                                    COLORWELL_SQUARE_WELL);
                        status = B_OK;
                  }
            }
            else
            if (msg->what == B_GET_PROPERTY)
            {
                  reply.AddBool("result",
                              Style() == COLORWELL_ROUND_WELL);
                  status = B_OK;
            }
            break;
      }
      case 1: // Red
      {
            if (msg->what == B_SET_PROPERTY)
            {
                  int32 newValue;
                  if (msg->FindInt32("data", &newValue) == B_OK)
                  {
                        rgb_color color = ValueAsColor();
                        color.red = newValue;
                        SetValue(color);
                        status = B_OK;
                  }
            }
            else
            if (msg->what == B_GET_PROPERTY)
            {
                  rgb_color color = ValueAsColor();
                  reply.AddInt32("result", color.red);
                  status = B_OK;
            }
```

```
			break;
		}
		case 2: // Green
		{
			if (msg->what == B_SET_PROPERTY)
			{
				int32 newValue;
				if (msg->FindInt32("data", &newValue) == B_OK)
				{
					rgb_color color = ValueAsColor();
					color.green = newValue;
					SetValue(color);
					status = B_OK;
				}
			}
			else
			if (msg->what == B_GET_PROPERTY)
			{
				rgb_color color = ValueAsColor();
				reply.AddInt32("result", color.green);
				status = B_OK;
			}
			break;
		}
		case 3: // Blue
		{
			if (msg->what == B_SET_PROPERTY)
			{
				int32 newValue;
				if (msg->FindInt32("data", &newValue) == B_OK)
				{
					rgb_color color = ValueAsColor();
					color.blue = newValue;
					SetValue(color);
					status = B_OK;
				}
			}
			else
			if (msg->what == B_GET_PROPERTY)
			{
				rgb_color color = ValueAsColor();
				reply.AddInt32("result", color.blue);
				status = B_OK;
			}
			break;
		}
		default:
			return BControl::MessageReceived(msg);
}

// If we were not able to handle one of our messages, we have
// an error condition. We need to describe the error using strerror
// and make it clear that something went wrong.
if (status != B_OK)
{
	reply.what = B_MESSAGE_NOT_UNDERSTOOD;
	reply.AddString("message", strerror(status));
}
```

```
	// We return an error code even if we were successful.
	reply.AddInt32("error", status);

	msg->SendReply(&reply);
}
```

`ResolveSpecifier()` can be short and simple or long and more complicated depending on the kinds of properties that your control uses. The purpose of this method is to determine which handler is supposed to receive and respond to the message.

```
BHandler *  ResolveSpecifier(BMessage *msg, int32 index,
                        BMessage *specifier, int32 what,
                        const char *property);
```

`msg` points to the scripting message being passed around. `specifier` contains the current specifier which is found at the index `index`. `what` holds the `what` value of the `specifier` message and `property` contains the name of the targeted property.

There are a few different ways that your properties can respond to a scripting message, and the amount of code involved in implementing `ResolveSpecifier()` is a direct result of these actions. The first way is for a property to return a BHandler which is attached to some other BLooper. The second is when a property returns a BHandler which is attached to the same BLooper as your control. The third is to return a value which is handled by your control, such as a calculated value or the result of calling your control's methods.

ResolveSpecifier() Method #1: Handler in a Remote Looper

BApplication uses this method to resolve its `Window` property. This is an excerpt from Haiku's source code for BApplication. Here the BApplication is handling a specifier for a window by index or reverse index.

```
if (propInfo.FindMatch(message, 0, specifier, what, property, &data) >= 0)
{
	switch (data) {
		case kWindowByIndex:
		{
			int32 index;
			err = specifier->FindInt32("index", &index);
			if (err != B_OK)
				break;

			if (what == B_REVERSE_INDEX_SPECIFIER)
				index = CountWindows() - index;

			BWindow *window = WindowAt(index);
			if (window != NULL) {
				message->PopSpecifier();
				BMessenger(window).SendMessage(message);
			} else
				err = B_BAD_INDEX;
			break;
		}
	}
}
```

At the end of BApplication's version of this function is a call to return `NULL`. The key here is the call to `PopSpecifier()` and the `NULL` return value. The scripting message is passed on to the proper messenger, but the specifier is popped off so that the target doesn't try to resolve the same one, and `NULL` is returned by the function because the BLooper is no longer responsible to resolve the specifier – the targeted BLooper will take it from here.

ResolveSpecifier() Method #2: Handler in the Current Looper

BView takes this route to resolve its `View` property. If the view has children, they will obviously belong to the same looper.

```
case 4: // View property
{
	if (!fFirstChild) {
		err = B_NAME_NOT_FOUND;
		replyMsg.AddString("message", "This window doesn't have "
			"children.");
		break;
	}

	// Get the child view based on what method was used
	BView* child = NULL;
	switch (what) {
		case B_INDEX_SPECIFIER: {
			int32 index;
			err = specifier->FindInt32("index", &index);
			if (err == B_OK)
				child = ChildAt(index);
			break;
		}
		case B_REVERSE_INDEX_SPECIFIER: {
			int32 rindex;
			err = specifier->FindInt32("index", &rindex);
			if (err == B_OK)
				child = ChildAt(CountChildren() - rindex);
			break;
		}
		case B_NAME_SPECIFIER: {
			const char* name;
			err = specifier->FindString("name", &name);
			if (err == B_OK)
				child = FindView(name);
			break;
		}
	}

	// Pass the message to the proper child...
	if (child != NULL) {
		msg->PopSpecifier();
		return child;
	}

	// ...or not, if it wasn't found
	if (err == B_OK)
		err = B_BAD_INDEX;

	replyMsg.AddString("message",
		"Cannot find view at/with specified index/name.");
```

```
		break;
}
```

While this way also calls `PopSpecifier()` to avoid repeatedly processing the same specifier, it returns a non-`NULL` value because the BHandler targeted belongs to the same BLooper that initiated the specifier resolution.

ResolveSpecifier() Method #3: Resolution

Most specifiers will be resolved by the target that receives them, and this is, by far, the most common way of resolving specifiers. In these instances, your control will return itself. If it isn't able to resolve everything, the control should return the parent class' version of `ResolveSpecifier()`.

```
BHandler *
ColorWell::ResolveSpecifier(BMessage *msg, int32 index,
                            BMessage *specifier, int32 what,
                            const char *property)
{
	BPropertyInfo propertyInfo(sColorWellProperties);
	int32 index = propertyInfo.FindMatch(msg, index, specifier, what,
                                         property);
	// If the property happens to be one in ColorWell's list, return
	// the ColorWell object.
	if (index >= 0)
		return this;

	// If we make it this far, it means that it's not one we recognize,
	// so we will return the inherited version.
	return BControl::ResolveSpecifier(msg, index, specifier, what,
                                      property);
}
```

Concluding Thoughts

Having finished with all three scripting-related functions, we almost have a completely finished control. Sending messages to it via `hey` works the way it should, so now it's possible to change its color remotely. What could possibly be left? Only a little, as we'll see in the next lesson.

Going Further

- Setting each color channel individually is a lot of work. Come up with a way to set all three at once.
- Create properties to set the color using the HSL (Hue, Saturation, Luminance) color model.