Copyright ©1996 by NeXT Software, Inc. All Rights Reserved.

Chapter 5

Debugging Tips

Once you have finished running the required conversion scripts and you have run any optional conversions that you'd like, debug your program. The conversion scripts are not totally automatic, and you will find bugs in your application after running them. This chapter provides some tips for debugging your newly converted program.

Before you debug, convert your nib files as described in the first chapter of this guide. After you convert your nib files, build your program. Be sure that you have added **AppKit.framework** and **Foundation.framework** and saved your project before you build.

Debugging Object Allocation and Deallocation

It's likely that object allocation and deallocation will cause most of the new problems in your application. Two common problems are using an object after it's been deallocated and releasing an object too many times.

You may want to debug the rest of your program first, saving the release problems until later. The **enableRelease:** convenience method defined in the Foundation NSAutoreleasePool class helps you ignore autorelease errors. NSAutoreleasePool defines the application's autorelease pool. When an object is autoreleased, it is added to the autorelease pool. At the top of the event loop, all objects in the pool are sent a **release** message, which decrements the reference count and potentially deallocates the object. NSAutoreleasePool allows you to control that pool.

If you receive messages from the debugger indicating that you are sending messages to deallocated objects,

enter this command:

```
(gdb) call [NSAutoreleasePool enableRelease:NO]
```

This message disables the deallocation of all autoreleased objects in your program. (It does not affect objects to which you send **release** directly.) You now can debug the rest of your program ignoring autorelease errors.

When you are ready to debug the autorelease errors, there are several ways you can go about it. All of them make your program run much slower than normal.

Debugging Autorelease Errors in gdb

If you are releasing an object too many times, enter these commands while your program is running in **gdb**:

```
(gdb) call [NSAutoreleasePool enableFreedObjectCheck:YES]
(gdb) break _NSAutoreleaseFreedObject
```

After you use **enableFreedObjectCheck:**, all **autorelease** and **release** messages first check to see if the object is already in an autorelease pool. If it is, they won't deallocate the object.

When the program hits the breakpoint, you can do a **backtrace** command to see what method was releasing the object. If you do an **up** at each stack frame until you get back to your own code, you can see the actual line where an object was released.

Using the oh Command

Another way to debug the **autorelease** and **release** errors is to use the **oh** command in conjunction with **gdb**.

1. Set this environment variable:

   ```
   % setenv NSZombieEnabled YES
   ```

   When this variable is set, the memory for deallocated objects is not reclaimed. Released objects are instead turned into ªzombies.º The advantage to setting this variable is that you can ensure that an object's address is unique.

2. Start your application in **gdb**.

3. From a different Terminal window, use **ps** to find out the ID of the process you are debugging and then perform this command:

   `% oh` *pid* `start`

   The **start** option tells **oh** to start recording allocation and deallocation events.

4. Go back to the first Terminal window and continue debugging your program in **gdb**. When you receive an autorelease error, it tells you the address of the object that is being released twice.

5. When you receive an autorelease error, perform the following command in the second Terminal window:

   `% oh` *pid address*

   where *address* is address of the object that is being released twice. **oh** will produce a report showing you the stack frame each time that object is allocated, copied, retained, or released, like the one shown below.

```
== Stacks for address 0x332114, in temporal order (oldest first):
(
    _NSAllocateObject,
    "+[NSNotification notificationWithName:object:]",
    "-[NSWindow _postWindowNeedsDisplay]",
    "-[NSWindow setViewsNeedDisplay:]",
    "-[NSView _setWindow:]",
    "-[NSView addSubview:]",
    "-[NSImage _focusOnCache:]",
    "-[NSImage _cacheRepresentation:stayFocused:]",
    "-[NSImage _lockFocusOnRep:]",
    "-[NSImage lockFocus]",
    "-[NSCursor set]",
    "-[NSApplication sendEvent:]",
```

```
    __NXFinishActivation,
    __NXDoDelayedWindowOrdering,
    "-[NSWindow sendEvent:]",
    "-[NSApplication sendEvent:]",
    "-[NSApplication run]",
    _main,
    start
)
(
    "-[NSObject autorelease]",
    "+[NSNotification notificationWithName:object:]",
    "-[NSWindow _postWindowNeedsDisplay]",
    "-[NSWindow setViewsNeedDisplay:]",
    "-[NSView _setWindow:]",
    "-[NSView addSubview:]",
    "-[NSImage _focusOnCache:]",
    "-[NSImage _cacheRepresentation:stayFocused:]",
    "-[NSImage _lockFocusOnRep:]",
    "-[NSImage lockFocus]",
    "-[NSCursor set]",
    "-[NSApplication sendEvent:]",
    __NXFinishActivation,
    __NXDoDelayedWindowOrdering,
    "-[NSWindow sendEvent:]",
    "-[NSApplication sendEvent:]",
    "-[NSApplication run]",
    _main,
    start
)
```

Keeping Memory Allocation Statistics

Another command, **AnalyzeAllocation**, lets you look at memory allocation after your program has finished executing. To use **AnalyzeAllocation**:

1. Set this environment variable:

   ```
   % setenv NSKeepAllocationStatistics YES
   ```

   The **NSKeepAllocationStatistics** variable tells your program to record information about memory allocation in a file named **/tmp/alloc_stats**_*name_pid*.

2. Run a specific task in your application.The allocation statistics file becomes very large very quickly, so it is important not to run too much of your program at once with **NSKeepAllocationStatistics** turned on.

3. Turn off the environment variable:

   ```
   % unsetenv NSKeepAllocationStatistics
   ```

4. Perform this command in a Terminal window:

   ```
   % AnalyzeAllocation -v /tmp/alloc_stats_name_pid
   ```

Common Autorelease Mistakes

Once you find the object with the autorelease error, look for the following:

592373_SquareBullet.eps ¬ For every **autorelease** and **release** message in your application, make sure there is a corresponding **alloc**, **copy**, **mutableCopy**, or **retain** message sent to the same object. As stated in Chapter€Chapter 1, **autorelease** and **release** decrement an object's reference count. **alloc**, **copy**, **mutableCopy**, and **retain** increment the reference count. The number of increments and decrements for an object must be equal. Another way of thinking about this is: If you don't allocate, copy, or retain an object, you're not responsible for releasing it.

592373_SquareBullet.eps ¬ When an NSArray, NSDictionary, or NSSet (known as the collection classes) is deallocated, the objects stored in the collection are released as well. If you need to access an object you stored in a collection after the collection is released, you must retain that object before you release the

collection.

592373_SquareBullet.eps ¬ Superviews retain subviews as you add them to the hierarchy and release subviews as you remove them from the hierarchy. If you swap views in and out of the hierarchy, you should retain the views that are not in the hierarchy.

592373_SquareBullet.eps ¬ When you change a window's content view, the window releases the old content view and retains the new content view.

592373_SquareBullet.eps ¬ Objects do not retain their delegates (to avoid retain cycles).

592373_SquareBullet.eps ¬ **decodeValuesOfObjCTypes:** returns a retained object. **decodeObject** returns an autoreleased object. If you unarchive an instance variable with **decodeObject**, send it the **retain** message.

592373_SquareBullet.eps ¬ You don't have to release an object unless you've explicitly allocated, copied, or retained it. Some methods in previous versions returned an object that you had to free. For example, Matrix's **selectedCells** method returned a List that the receiving object had to free. If you used such a method, look for unnecessary releases in your code.

For more information about object allocation and deallocation, see ªNSObject Conversionº in the chapter ªGlobal API and Style Changes.º

Trouble Loading a Nib File

If your application can't load a nib file, it may be one of two problems:

592373_SquareBullet.eps ¬ Your nib file may not have been converted properly. Try opening it in Release 4.0 Interface Builder. When you do so, Interface Builder automatically converts the nib file.

592373_SquareBullet.eps ¬ If you still have problems after opening the nib in Interface Builder, there may be an error in one of your **initWithCoder:** methods. In particular, if you archive a color, make sure you are invoking the correct method. Use **decodeNXColor** to retrieve an old NXColor structure. Use **decodeObject** to

retrieve an NSColor object. Remember that **decodeObject** requires that you retain the object, **decodeValuesOfObjCTypes:** does not. For more complete information on unarchiving, see ªArchiver Conversionº in ªConverting the Common Classes.º

description Methods

The debugger's **print-object** command (**po**) invokes the object's **description** method. Every OpenStep object responds to this method. The default **description**, implemented in NSObject, simply gives the object's class and address. Some classes implement different descriptions. For example, performing **po** on an NSString prints out the string. NSArrays and NSDictionary objects print out the descriptions of each of their elements. You might find it useful to implement a **description** method for some of your classes.

Exceptions

If your program raises an exception that you don't recognize, set a breakpoint on the **raise** method in NSException. All exceptions are raised using this method. When the program encounters this breakpoint, use the **backtrace** command to find where the exception is being raised.

Keeping Notification Statistics

You can have your program keeps notification statistics, similar to the way it keeps memory allocation statistics. To keep notification statistics, set this environment variable

```
% setenv NSKeepNotificationStatistics YES
```

and then run your program. Information about observer objects added to the notification center, observer objects removed from the notification center, and notifications sent are recorded in the file **/tmp/note_stats**_*name_pid*. One entry in this file looks like this:

```
-170202445.213512 bffff570 1800d039 ADD NSNotificationCenter fe738 _bundleLoaded:
NSBundleLoaded Nil 0
```

This entry gives you the time at which the event occurred (in seconds relative to the absolute reference date), the stack frame, the program counter, and what type of event it was: **ADD**, which means an observer was added to the notification center, **POST**, which means a notification was posted, and **SUB**, which means an observer was removed. If your program invokes:

```
[[NSNotificationCenter defaultCenter] addObserver:self
     selector:@selector(windowMoved:)
     name:NSWindowDidMoveNotification
     object:importantWindow];
```

the log records an **ADD** event with the class and address of the observer added (**self**), the selector to be invoked (**windowMoved:**), the name of the notification to be observed (**NSWindowDidMoveNotification**), and the class and address of the object that will post the notification (NSWindow).

For **POST** events:

```
[[NSNotificationCenter defaultCenter]
     postNotificationName:aNotification
     object:self
     userInfo:aDictionary];
     object:importantWindow];
```

the log records the notification name (**aNotification**) and the number of observers the notification was sent to.

For **SUB** events:

```
[[NSNotificationCenter defaultCenter] removeObserver:self
     name:NSWindowDidMoveNotification
     object:importantWindow];
```

the log records the class name and address of the observer (**self**), the notification no longer being observed (**NSWindowDidMoveNotification**), and the class and address of the object that sends the notification (NSWindow).

Gotchas

This section lists a few tricky areas the compiler does not flag. Each of these areas has been described in more detail in earlier chapters in this guide. They are listed here as a convenience. Each of the sections below provides a reference to the chapter in this guide where you can find more information on the subject. If you need still more information, see the *Foundation Framework Reference*, the *Application Kit Reference*, and the *OpenStep Specification*.

Dereferencing Objects

Many items that were previously structures, for example events and colors, are now objects. You might have used the **&** operator to pass the address of one of these structures to a method or function. Now that the structures have become objects, the **&** operator returns an invalid address.

NSStrings

For NSString, **nil** represents an invalid value and **@""** represents an empty string. In general, you should use **@""** where you used to use NULL. For example, you should never send **setStringValue:nil**; always **setStringValue:@""**. The message **setStringValue:nil** produces a run-time error.

For more information about NSStrings, see the chapter ªConverting the Common Classes.º

NSPopUpButtons

If you sent **target** to an NSPopUpButton, remember that this is now the target of the list rather than the target of the trigger button. You should change the target in the nib file accordingly.

For more information about NSPopUpButtons, see the chapter ªConverting Application Kit Classes.º

Obtaining Icons

If your application accesses a named image that it cannot find but the image exists in the MachO section, you will receive an error. If you are using **imageNamed:** to retrieve the icon, you should store the icon in the application's wrapper.

For more information about images and icons, see the chapter ªConverting Application Kit Classes.º

Background Colors

Your code might use the **setBackgroundGray:** method to disable the background by passing it a value of -1.0. In OpenStep, gray values and color values are no longer separate, so the **setBackgroundGray:** method is obsolete. The conversion process changed it to **setBackgroundColor:**. Because you are passing **setBackgroundColor:** the value -1.0, an illegal color value, you receive an error. Use **setDrawsBackground:NO** instead to disable the background.

For more information on working with color, see ªConverting Application Kit Classes.º

Giving up First Responder Status

The NSText delegate method **textShouldEndEditing:** returns YES if the NSText object should end editingÐthat is, it should give up its first responder status. The NEXTSTEP delegate method that **textShouldEndEditing:** replaces returned YES if the Text object should *not* give up first responder status.

NeXT's implementation of OpenStep provides a keyboard interface feature that allows users to use the keyboard instead of the mouse. Because of the way this feature works, first responder might be taken from objects when the user clicks in a button in the same window. To implement an object that takes advantage of this feature, you must implement **acceptsFirstResponder** rather than rely on receiving an initial **makeFirstResponder** message.

For more information on the NSText object, see ªConverting Application Kit Classes.º