# Dynamic Library Programming Topics

# Contents

## Creating Dynamic Libraries   49

## Using Dynamic Libraries   67

## Logging Dynamic Loader Events   79

## Document Revision History   81

# Figures, Tables, and Listings

## Creating Dynamic Libraries   49

## Using Dynamic Libraries   67

## Logging Dynamic Loader Events   79

**6**

# Introduction to Dynamic Library Programming Topics

Applications are rarely implemented as a single module of code because operating systems implement much of the functionality applications need in libraries. To develop applications, programmers link their custom code against these libraries to get basic functionality, such as the ability to write to standard output or draw complex images using a graphics card. However, linking to libraries creates large executable files and wastes memory. One way to reduce the file size and memory footprint of applications is to reduce the amount of code that is loaded at application launch. Dynamic libraries address this need; they can be loaded either at application launch time or at runtime. Dynamic libraries provide a way for applications to load code when it's actually needed.

To load dynamic libraries at runtime, applications should use a set of efficient and portable functions, called dynamic loader compatibility functions. Using these functions ensures that dynamic libraries are loaded in the most efficient way and facilitates the porting of applications from one platform to another.

This document is intended for developers of dynamic libraries and developers who use dynamic libraries in their applications. You should be familiar with the Mac OS, UNIX, Solaris, or Linux operating systems. You should also be an experienced C, C++, or Objective-C programmer.

This document explains how dynamic libraries are loaded at application launch time and how to use the DLC functions, `dlopen`, `dlsym`, `dladdr`, `dlclose`, and `dlerror`, to load dynamic libraries at runtime. This document also provides guidelines for developing dynamic libraries to be used by client applications.

This document doesn't address the needs of Carbon or Cocoa programmers who need to load code packaged in a framework or a bundle at runtime or those who want to learn how to package dynamic libraries into frameworks or bundles. The documents *Framework Programming Guide*, and *Code Loading Programming Topics for Cocoa* provide information tailored specifically to Carbon and Cocoa developers.

After reading this document, you will understand how dynamic libraries should be implemented so they can be used effectively by client applications. You will also know how to use the cross-platform DLC functions to load dynamic libraries at runtime.

## Organization of This Document

This document is comprised by the following articles:

- "Overview of Dynamic Libraries" (page 9) explains what dynamic libraries are and how they are used by applications. It also provides an overview of the Mac OS X dynamic loader and its cross-platform programming interface.

- "Dynamic Library Design Guidelines" (page 15) explains how to name and export public symbols so that they are easy to use by clients and how to implement static initializers and finalizers. This article also provides tips on how to manage compatibility and dependency issues. Finally, this article lists issues to consider when implementing a dynamic library using C++ or Objective-C.

- "Dynamic Library Usage Guidelines" (page 35) shows how to correctly load and use dynamic libraries in applications or in other dynamic libraries. It also provides specific details for using C++–based libraries and Objective-C–based libraries.

- "Creating Dynamic Libraries" (page 49) demonstrates how to develop dynamic libraries so that they are easy to use by developers who want to take advantage of them in their products.

- "Using Dynamic Libraries" (page 67) describes the process of installing and using dynamic libraries. It also shows how to interpose the functions exported by a dynamic library.

- "Logging Dynamic Loader Events" (page 79) explains how to turn on dynamic loader logging and identifies the events that can be logged.

This document also includes a revision history.

# See Also

To complement the information provided in this document, consult the following documents:

- *Code Size Performance Guidelines* focuses on techniques you can use to reduce the size of your code and use memory efficiently by grouping code into modules according to functionality.

- "Executing Mach-O Files" in *Mach-O Programming Topics* describes the application-launch process. In particular this article explains how the dynamic loader binds imported symbols and explains the advantages of using a two-level namespace instead of a flat namespace. This article also explains the different storage classes and symbol attributes you can use to control the visibility of a dynamic library's exported symbols.

- "Loading Code at Runtime" in *Mach-O Programming Topics* explains how to package a dynamic library as a framework, which provides features such as resource management and a streamlined versioning model.

You can find further information on dynamic libraries in the following documents:

- *Framework Programming Guide* explains how to package dynamic libraries into frameworks.

- *Porting UNIX/Linux Applications to Mac OS X* explains how to use dynamic libraries (plug-ins) to reduce the amount of platform-specific code in an application.

- *How to Write Shared Libraries* (http://people.redhat.com/drepper/dsohowto.pdf) describes in great detail Linux's shared objects (dynamic libraries) and provides guidelines on the design and implementation of a library's API and ABI, including compatibility, performance, and security issues.

# Overview of Dynamic Libraries

Two important factors that determine the performance of applications are their launch times and their memory footprints. Reducing the size of an application's executable file and minimizing its use of memory once it's launched make the application launch faster and use less memory once it's launched. Using dynamic libraries instead of static libraries reduces the executable file size of an application. They also allow applications to delay loading libraries with special functionality only when they're needed instead of at launch time. This feature contributes further to reduced launch times and efficient memory use.

This article introduces dynamic libraries and shows how using dynamic libraries instead of static libraries reduces both the file size and initial memory footprint of the applications that use them. This article also provides an overview of the dynamic loader compatibility functions applications use to work with dynamic libraries at runtime.

## What Are Dynamic Libraries?

Most of an application's functionality is implemented in libraries of executable code. When an application is linked against a library using a static linker, the code that the application uses is copied to the generated executable file. The kind of library that becomes part of an application's executable file is known as a static library. Static libraries are collections or archives of object files.

> **Note:** Static libraries are also known as *static archive libraries* and *static linked shared libraries*.

When an application is launched, the application's code—which includes the code of the static libraries it was linked against—is loaded into the application's address space. Linking many static libraries into an application produces large application executable files. Figure 1 shows the memory usage of an application that uses functionality implemented in static libraries. Applications with large executables suffer from slow launch times and large memory footprints. Also, when a static library is updated, its client applications don't benefit from the improvements made to it. To gain access to the improved functionality, the application's developer must link the application's object files against the new version of the library. And the applications users would have to replace their copy of the application with the latest version. Therefore, keeping an application up to date with the latest functionality provided by static libraries requires disruptive work by both developers and end users.

**Figure 1**        Application using static libraries
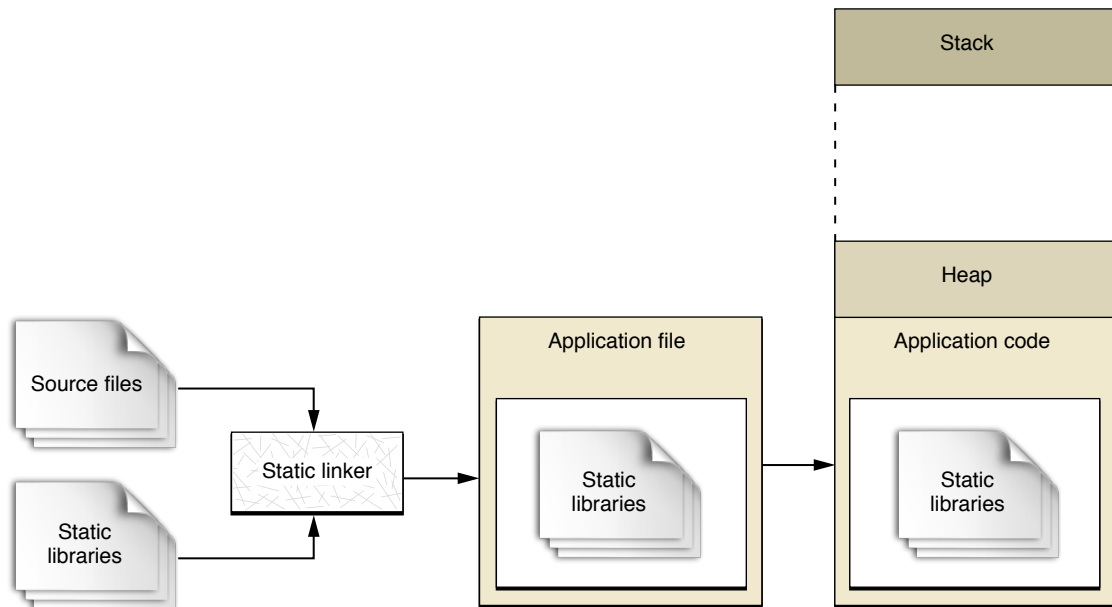


A better approach is for an application to load code into its address space when it's actually needed, either at launch time or at runtime. The type of library that provides this flexibility is called **dynamic library**. Dynamic libraries are not statically linked into client applications; they don't become part of the executable file. Instead, dynamic libraries can be loaded (and linked) into an application either when the application is launched or as it runs.

**Note:** Dynamic libraries are also know as *dynamic shared libraries*, *shared objects*, or *dynamically linked libraries*.

Figure 2 shows how implementing some functionality as dynamic libraries instead of as static libraries reduces the memory used by the application after launch.

**Figure 2**        Application using dynamic libraries
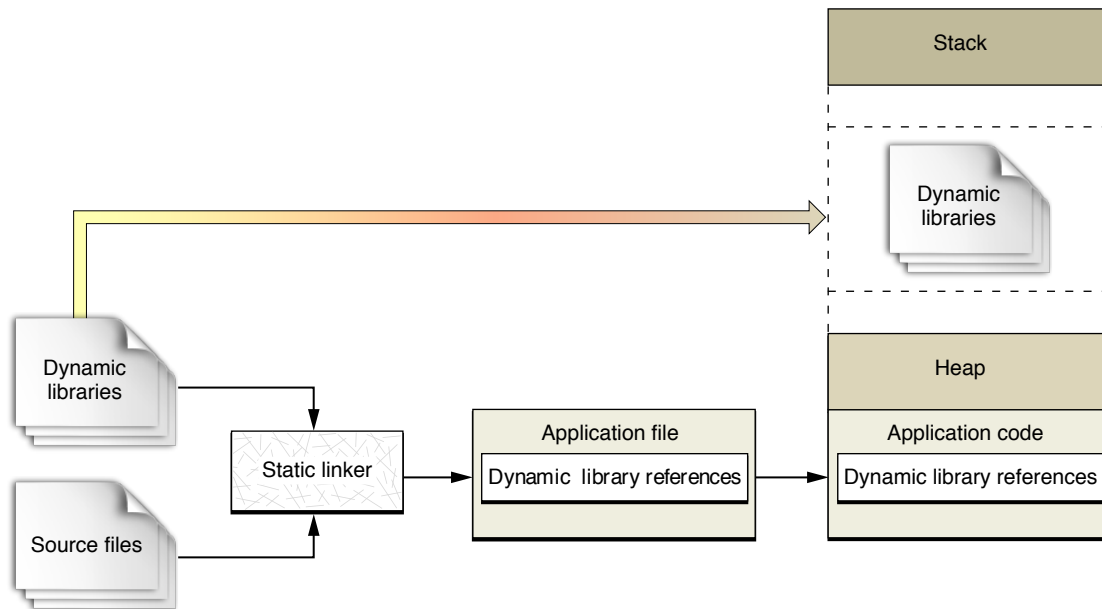


Using dynamic libraries, programs can benefit from improvements to the libraries they use automatically because their link to the libraries is dynamic, not static. That is, the functionality of the client applications can be improved and extended without requiring application developers to recompile the applications. Applications written for Mac OS X benefit from this feature because all system libraries in Mac OS X are dynamic libraries. This is how applications that use Carbon or Cocoa technologies benefit from improvements to Mac OS X.

Another benefit dynamic libraries offer is that they can be initialized when they are loaded and can perform clean-up tasks when the client application terminates normally. Static libraries don't have this feature. For details, see "Module Initializers and Finalizers" (page 25).

One issue that developers must keep in mind when developing dynamic libraries is maintaining compatibility with client applications as a library is updated. Because a library can be updated without the knowledge of the client-application's developer, the application must be able to use the new version of the library without changes to its code. To that end, the library's API should not change. However, there are times when improvements require API changes. In that case, the previous version of the library must remain in the user's computer for the client application to run properly. "Dynamic Library Design Guidelines" (page 15) explores the subject of managing compatibility with client applications as a dynamic library evolves.

# How Dynamic Libraries Are Used

When an application is launched, the Mac OS X kernel loads the application's code and data into the address space of a new process. The kernel also loads the dynamic loader ( /usr/lib/dyld ) into the process and passes control to it. The dynamic loader then loads the application's **dependent libraries**. These are the dynamic libraries the application was linked against. The static linker records the filenames of each of the dependent libraries at the time the application is linked. This filename is known as the dynamic library's **install name**. The dynamic loader uses the application's dependent

libraries' install names to locate them in the file system. If the dynamic loader doesn't find all the application's dependent libraries at launch time or if any of the libraries is not compatible with the application, the launch process is aborted. For more information on dependent-library compatibility, see "Managing Client Compatibility With Dependent Libraries" (page 16). Dynamic library developers can set a different install name for a library when they compile it using the `gcc -install_name` option. See the `gcc` man page for details.

The dynamic loader resolves only the undefined external symbols the application actually uses during the launch process. Other symbols remain unresolved until the application uses them. For details on the process the dynamic loader goes when an application is launched, see "Executing Mach-O Files" in *Mach-O Programming Topics*.

The dynamic loader—in addition to automatically loading dynamic libraries at launch time—loads dynamic libraries at runtime, at the application's request. That is, if an application doesn't require that a dynamic library be loaded when it launches, developers can choose to not link the application's object files against the dynamic library, and, instead, load the dynamic library only in the parts of the application that require it. Using dynamic libraries this way speeds up the launch process. Dynamic libraries loaded at runtime are known as **dynamically loaded libraries**. To load libraries at runtime, applications can use functions that interact with the dynamic loader for the platform under which they're running.

> **Note:** The target architecture of the client and the dynamic library must be the same. Otherwise, the dynamic loader doesn't load the library.

Different platforms implement their dynamic loaders differently. They may also have custom dynamic code-loading interfaces make code difficult to port across platforms. To facilitate porting an application from UNIX to Linux, for example, Jorge Acereda and Peter O'Gorman developed the **dynamic loader compatibility (DLC)** functions. They offer developers a standard, portable way to use dynamic libraries in their applications.

The DLC functions are declared in `/usr/include/dlfcn.h`. There are five of them:

■ `dlopen`: Opens a dynamic library. An application calls this function before using any of the library's exported symbols. If the dynamic library hasn't been opened by the current process, the library is loaded into the process's address space. The function returns a handle that's used to refer to the opened library in calls to `dlsym` and `dlclose`. This handle is known as the **dynamic library handle**. This function maintains a reference count that indicates the number of times the current process has used `dlopen` to open a particular dynamic library.

■ `dlsym`: Returns the address of a symbol exported by a dynamically loaded library. An application calls this function after obtaining a handle to the library through a call to `dlopen`. The `dlsym` function takes as parameters the handle returned by `dlopen` or a constant specifying the symbol search scope and the symbol name.

■ `dladdr`: Returns information on the address provided. If the address corresponds to a dynamically loaded library within the application's address space, this function returns information on the address. This information is returned in a `Dl_info` structure, which encapsulates the pathname of the dynamic library, the library's base address, and the address and value of the nearest symbol to the address provided. If no dynamic library is found at the address provided, the function returns no information.

■ `dlclose`: Closes a dynamically loaded library. This function takes as a parameter a handle returned by `dlopen`. When the reference count for that handle reaches 0, the library is unloaded from the current process's address space.

■ `dlerror`: Returns a string that describes an error condition encountered by the last call to `dlopen`, `dlsym`, or `dlclose`.

For more information on the DLC functions, see *Mac OS X ABI Dynamic Loader Reference*.

# Dynamic Library Design Guidelines

Dynamic libraries, in addition to grouping common functionality, help reduce an application's launch time. However, when designed improperly, dynamic libraries can degrade the performance of their clients. (A dynamic library **client** is an application or a library that either is linked against the library or loads the library at runtime. This document also uses the word **image** to refer to dynamic library clients.) Therefore, before creating a dynamic library, you must define its purpose and its intended use. Devising a small and effective interface to the library's functionality goes a long way towards facilitating its adoption in other libraries or applications.

This article addresses the main issues dynamic library developers face when designing and implementing dynamic libraries. The focus of this article is to show you how to design libraries in a way that facilitates their improvement through revisions and makes it easy for the libraries' users to correctly interact with the library.

## Designing an Optimal Dynamic Library

Dynamic libraries contain code that can be shared by multiple applications in a user's computer. Therefore, they should contain code that several applications can use. They should not contain code specific to one application. These are the attributes of an optimal dynamic library:

■ **Focused:** The library should focus on few, highly related goals. A highly focused library is easier to implement and use than a multi-purpose library.

■ **Easy to use:** The library's interface, the symbols that the clients of the library use to interact with it, should be few and easy to understand. A simple interface allows a library's users to understand its functionality faster than they could understand a large interface.

■ **Easy to maintain:** The library's developers must be able to make changes to the library that improve its performance and add features. A clear separation between a library's private and public interfaces gives library developers freedom to make profound changes to the library's inner workings with minimal impact to its clients. When designed properly, a client created with an early version of a library can use the latest version of the library unchanged and benefit from the improvements it provides.

# Managing Client Compatibility With Dependent Libraries

The client of a dynamic library can use the library in two ways: as a dependent library or as a runtime loaded library. A **dependent library**, from the client's point of view, is a dynamic library the client is linked against. Dependent libraries are loaded into the same process the client is being loaded into as part of its load process. For example, when an application is launched, its dependent libraries are loaded as part of the launch process, before the main function is executed. When a dynamic library is loaded into a running process, its dependent libraries are loaded into the process before control is passed to the routine that opened the library.

A **runtime loaded library** is a dynamic library the client opens with the `dlopen` function. Clients do not include runtime loaded libraries in their link line. The dynamic loader, therefore, doesn't open these libraries when the client is loaded. Clients open a runtime loaded library when they're about to use a symbol it exports. Clients don't have any undefined external references to symbols in runtime loaded libraries. Clients get the address of all the symbols they need from runtime loaded libraries by calling the `dlsym` function with the symbol name.

A client must always be compatible with its dependent libraries. Otherwise, an application doesn't launch or a runtime loaded library fails to load.

When you design a dynamic library, you may have to consider its ongoing maintenance. Sometime you may have to make changes to the library to implement new features or to correct problems. But you also have to think about the library's existing clients. There are two types of revisions you can make to a library: revisions that are compatible with current clients and require no client-developer intervention and revisions that require that clients be linked against the new revision. The former are **minor revisions** and the latter are **major revisions**.

The following sections explore compatibility issues to consider before implementing a library that may need to updated at a later time.

## Defining Client Compatibility

As a library upon which existing clients depend is revised, changes to it may affect the clients' ability to use new versions of the library. The degree to which a client can use earlier or later versions of a dependent library than the one it is linked against is called **client compatibility**.

Some changes are minor; these include adding symbols that are unknown to clients. Other changes are major; such changes include removing a symbol, changing a symbol's size or visibility, and changing the semantics of a function. All the symbols a library exposes to clients make up the library's ABI (application binary interface). The library's API (application programming interface) comprises only the functions that a library makes available to its clients. The degree to which a library's ABI remains the same from the point of view of the clients that were developed using an earlier version of the library determines the library's stability. Ensuring that a library's ABI remains stable guarantees that clients can use newer versions of the library unchanged. That is, users of an application that depends on a library that's updated regularly can see their application's performance improve as they update the library (think of the Mac OS X Software Update mechanism) without obtaining a new version of the application.

For example, assume an application is linked against the first version of a dynamic library and released to end users. Later, the library's developer makes minor changes to the library and releases it. When end users install the new version of the library on their computers, the application can to use the new

version without requiring the end users to get an updated application file from the developer. The application may benefit from efficiency improvements made to the library's API. And, depending on how the library was written, the application might be able to take advantage of features introduced by the new version. However, these features are accessed by the application only through the API available in the first version of the library. Any interfaces introduced in the new version of the library go unused by the application.

Figure 1 (page 17) illustrates the life cycle of the Draw dynamic library and one of its clients.

**Figure 1**      The life cycle of a dynamic library and a client
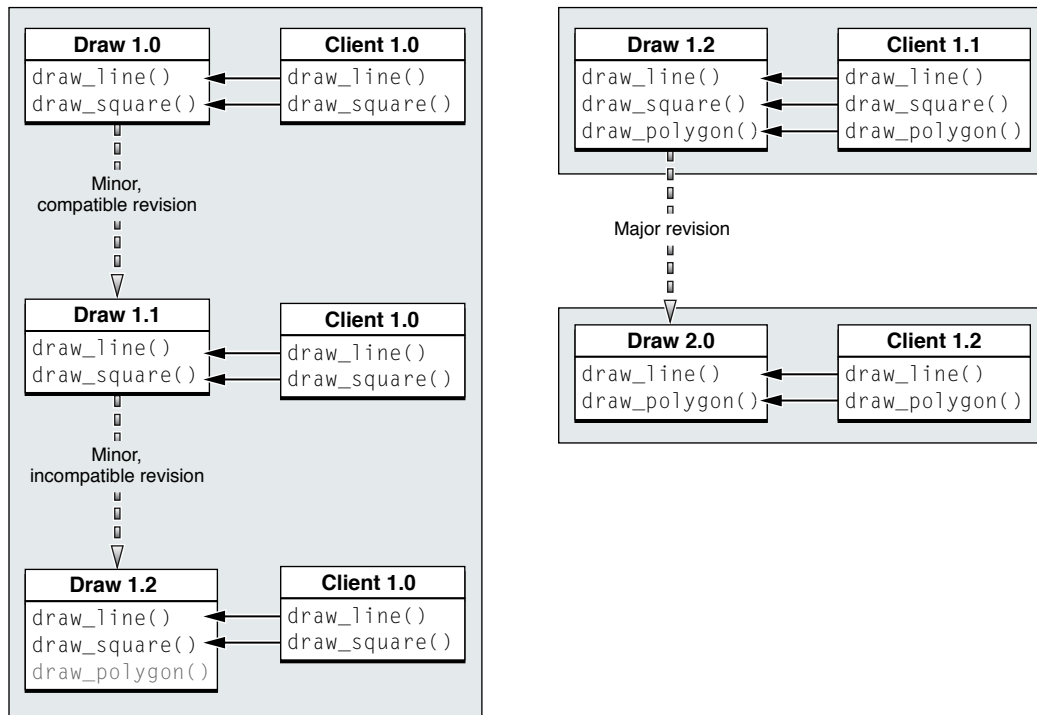


This list describes the versions of the library and the client:

■   Draw 1.0 is the initial version of the library. It exports two functions, `draw_line` and `draw_square`.

■   Client 1.0 is linked against Draw 1.0. Therefore, it can use the two symbols the library exports.

■   Draw 1.1 has faster versions of `draw_line` and `draw_square`, but their semantics are unchanged, maintaining client compatibility. This is a compatible, minor revision because Client 1.0 can use Draw 1.1.

■   Draw 1.2 introduces the `draw_polygon` function. The API of the new revision of the library is a superset of the previous version's API. The Draw 1.1 API subset of the 1.2 version is unchanged. Therefore, Client 1.0 can use Draw 1.2. However, Client 1.0 doesn't know of the existence of `draw_polygon` and, therefore, it doesn't use it. This is a minor revision because the API Client 1.0 knows about is unchanged in Draw 1.2. But this is also an incompatible revision because the API changed. Clients linked against this version of the library cannot use earlier versions.

■ Client 1.1 is linked against Draw 1.2 and uses `draw_polygon`. Client 1.1 cannot use earlier versions of the library because it uses `draw_polygon`, a function that isn't exported by those versions. However, if the library's developer adds the `weak_import` attribute to the symbol's definition, Client 1.1 would be able to use earlier versions of the library by ensuring that `draw_polygon` exists in its namespace before using it. If the symbol isn't defined, the client may use other means of performing the desired task, or it may not perform the task. See "Symbol Exporting Strategies" (page 20) for details.

■ Draw 2.0 doesn't export `draw_square`. This is a major revision because a symbol exported in the previous version of the library is not exported in this version. Clients linked against this version of the library cannot use earlier versions.

Clients should be able to use all the minor revisions to the library they're linked against without relinking. In general, to use a major revision of a library, the client must be linked against the new version. The client may also need to be changed to take advantage of new symbols, to adapt its use of symbols that have been modified, or to not use symbols that are not exported by the new revision.

> **Note:** The header files for your libraries should include only the symbols the libraries' clients should actually use. If clients use symbols other than the ones you specify, they limit the compatibility of their products with new or earlier versions of your libraries.

## Specifying Version Information

The filename of a dynamic libraries normally contains the library's name with the lib prefix and the `.dylib` extension. For example, a library called Dynamo would have the filename `libDynamo.dylib`. However, if a library may go through one or more revisions after it's released, its filename must include the major version number of the revision. Clients linked against libraries whose filename includes the major version number of the revision never use a new major revision of the library because major revisions are published under different filenames. This versioning model prevents clients from using library revisions whose API is incompatible with the API known to the clients.

When you publish a dynamic library intended to have future revisions, you must disclose the library's major version number in its filename. For example, the filename for the first version of the Draw library, introduced in "Defining Client Compatibility" (page 16), could be `libDraw.A.dylib`. The letter A specifies the major version number for the initial release. You can use any nomenclature for the major version. For example, the Draw library could also be named `libDraw.1.dylib`, or `libDraw.I.dylib`. The important thing is that the filenames of subsequent major revisions of the library have different (and preferably incremental) major version numbers. Continuing the Draw library example, a major revision to the library could be named `libDraw.B.dylib`, `libDraw.2.dylib`, or `libDraw.II.dylib`. Minor revisions to the library are released under the same filename used by the previous major revision.

In addition to the major version number, a library has a minor version number. The minor version number is an incremental number using the format `X[.Y[.Z]]`, where X is a number between 0 and 65535, and Y and Z are numbers between 0 and 255. For example, the minor version number for the first release of the Draw library could be 1.0. To set the minor version number of a dynamic library, use the `gcc -current_version` option.

The compatibility version number is similar to the minor version number; it's set through the `gcc -compatibility_version` option. The compatibility version number of a library release specifies the earliest minor version of the clients linked against that release can use. For instance, the example in "Defining Client Compatibility" (page 16) indicates that Client 1.1 cannot use versions of the Draw library earlier than 1.2 because they don't export the `draw_polygon` function.

Before loading a dynamic library, the dynamic loader compares the current version of the `.dylib` file in the user's file system with the compatibility version of the `.dylib` file the client was linked against in the developer's file system. If the current version is earlier (less) than the compatibility version, the dependent library is not loaded. Therefore, the launch process (for client applications) or the load process (for client libraries) is aborted.

> **Note:** The dynamic loader performs the version compatibility test only with dependent libraries. Dynamic libraries opened at runtime with `dlopen` don't go through this test.

# Specifying Your Library's Interface

The most important aspect to define before implementing a dynamic library is its interface to its clients. The public interface affects several areas in the use of the library by its clients, the library's development and maintenance, and the performance of the applications in which the library is used:

- **Ease of use:** A library with a few but easily understandable public symbols is far easier to use than one that exports all the symbols it defines.

- **Ease of maintenance:** A library that has a small set of public symbols and an adequate set of private symbols, is far easier to maintain because there are few client entry points to test. Also, developers can change the private symbols to improve the library in newer versions without impacting the functionality of clients that were linked against an earlier version.

- **Performance:** Designing a dynamic library so that it exports the minimum number of symbols optimizes the amount of time the dynamic loader takes to load the library into a process. The fewer exported symbols a library has, the faster the dynamic loader loads it.

The following sections show how to determine which of the library's symbols to export, how to name them, and how to export them.

## Deciding What Symbols to Export

Reducing the set of symbols your library exports makes the library easy to use and easy to maintain. With a reduced symbol set, the users of your library are exposed only to the symbols that are relevant to them. And with few public symbols, you are free to make substantial changes to the internal interfaces, such as adding or removing internal symbols that do not affect clients of your library.

Global variables should never be exported. Providing uncontrolled access to a library's global variables leaves the library open to problems caused by clients assigning inappropriate values to those variables. It's also difficult to make changes to global variables from one version of your library to another without making newer revisions incompatible with clients that were not linked against them. One of the main features of dynamic libraries is the fact that, when implemented correctly, clients can use newer versions of them without relinking. If clients need to access a value stored in a global variable,

your library should export accessor functions but not the global variable itself. Adhering to this guideline allows library developers to change the definitions of global variables between versions of the library, without introducing incompatible revisions.

If your library needs the functionality implemented by functions it exports, you should consider implementing internal versions of the functions, adding wrapper functions to them, and exporting the wrappers. For example, your library may have a function whose arguments must be validated, but you're certain that the library always provides valid values when invoking the function. The internal version of the function could be optimized by removing validation code from it, making internal use more efficient. The validation code can then be placed in the wrapper function, maintaining the validation process for clients. In addition, you can further change the internal implementation of the function to include more parameters, for example, while maintaining the external version the same.

Having wrapper functions call internal versions reduces the performance of an application, especially if the function is called repeatedly by clients. However, the advantages of flexible maintenance for you and a stable interface for your clients greatly outweigh this negligible performance impact.

## Naming Exported Symbols

The dynamic loader doesn't detect naming conflicts between the symbols exported by the dynamic libraries it loads. When a client contains a reference to a symbol that two or more of its dependent libraries export, the dynamic loader binds the reference to the first dependent library that exports the symbol in the client's dependent library list. The **dependent library list** is a list of the client's dependent libraries in the order they were specified when the client was linked against them. Also, when the `dlsym` function is invoked, the dynamic loader returns the address of the first symbol it finds in the specified scope (global, local, or next) with a matching name. For details on symbol-search scope, see "Using Symbols" (page 39).

To ensure that your library's clients always have access to the symbols your library exports, the symbols must have unique names in a process's namespace. One way is for applications to use two-level namespaces. Another is to add prefixes to every exported symbol. This is the convention used by most of the Mac OS X frameworks, such as Carbon and Cocoa. For more information on two-level namespace, see "Executing Mach-O Files" in *Mach-O Programming Topics*.

## Symbol Exporting Strategies

After you have identified the symbols you want to expose to your library's users, you must devise a strategy for exporting them or for not exporting the rest of the symbols. This process is also known as setting the **visibility** of the symbols—that is, whether they are accessible to clients. Public or exported symbols are accessible to clients; private, hidden, or unexported symbols are not accessible to clients. In Mac OS X, there are several ways of specifying the visibility of a library's symbols:

■ The `static` storage class: This is the easiest way to indicate that you don't want to export a symbol.

■ The exported symbols list or the unexported symbols list: The list is a file with the names of symbols to export or a list of symbols to keep private. The symbol names must include the underscore (_) prefix. You can use only one type of list when generating the dynamic library file.

- The GCC `visibility` attribute: You place this attribute in the definition of symbols in implementation files to set the visibility of symbols individually. It gives you more granular control over which symbols are public or private.

- The `gcc -fvisibility` option: This option specifies at compilation time the visibility of symbols with unspecified visibility in implementation files. This option, combined with the `visibility` attribute, is the most safe and convenient way of identifying public symbols.

- The GCC `weak_import` attribute: Placing this attribute in the declaration of a symbol in a header file tells the compiler to generate a weak reference to the symbol. This feature is called **weak linking**; symbols with the `weak_import` attribute are called **weakly linked symbols**. With weak linking, clients do not fail to launch when the version of the dependent library found at launch time or load time doesn't export a weakly linked symbol referenced by the client. It's important to place the `weak_import` attribute in the header files that the source files of the library's clients use, so that the client developers know that they must ensure the existence of the symbol before using it. Otherwise, the client would crash or function incorrectly when it attempts to use the symbol. See "Using Weakly Linked Symbols" (page 42) for further details on weakly linked symbols. For more information on symbol definitions, see Executing Mach-O Files in *Mach-O Programming Topics*.

- The `gcc -weak_library` option: This option tells the compiler to treat all the library's exported symbols as weakly linked symbols.

To illustrate how to set the visibility of a library's symbols, let's start with a dynamic library that allows its clients to set a value kept in a global variable in the library, and to retrieve the value. Listing 1 shows the code that makes up the library.

**Listing 1**     A simple dynamic library

```
/* File: Person.h */
char* name(void);
void set_name(char* name);

/* File: Person.c */
#include "Person.h"
#include <string.h>
char _person_name[30] = {'\0'};
char* name(void) {
    return _person_name;
}

void _set_name(char* name) {
   strcpy(_person_name, name);
}

void set_name(char* name) {
    if (name == NULL) {
        _set_name("");
    }
    else {
        _set_name(name);
    }
}
```

The intent of the library's developer is to provide clients the ability to set the value of _person_name with the set_name function and to let them obtain the value of the variable with the name function. However, the library exports more than the name and set_name functions, as shown by the output of the nm command-line tool:

```
% gcc -dynamiclib Person.c -o libPerson.dylib
% nm -gm libPerson.dylib
...
libPerson.dylib(ccfM2qAA.o):
00001004 (__DATA,__data) external __person_name   // Inadvertently exported
00000eec (__TEXT,__text) external __set_name      // Inadvertently exported
00000eb8 (__TEXT,__text) external _name
00000f30 (__TEXT,__text) external _set_name
...
```

Note that the _person_name global variable and the _set_name function are exported along with the name and set_name functions. There are many options to remove _person_name and _set_name from the symbols exported by the library. This section explores a few.

The first option is to add the static storage class to the definition of _person_name and _set_name in Person.c , as shown in Listing 2.

**Listing 2**      Person module hiding a symbol with the static storage class

```
/* File: Person.c */
#include "Person.h"
#include <string.h>

static char _person_name[30] = {'\0'};         // Added 'static' storage class
char* name(void) {
    return _person_name;
}

static void _set_name(char* name) {            // Added 'static' storage class
   strcpy(_person_name, name);
}

void set_name(char* name) {
    if (name == NULL) {
        _set_name("");
    }
    else {
        _set_name(name);
    }
}
```

Now, the nm output, looks like this:

```
libPerson.dylib(ccPQ6JXF.o):
00000ebc (__TEXT,__text) external _name
00000f34 (__TEXT,__text) external _set_name
```

This means that the library exports only name and set_name. Actually, the library also exports some undefined symbols, including strcpy. They are references to symbols the library obtains from its dependent libraries.

> **Note:** You should always use the static storage class for symbols that you want to keep private for a specific file. It's a very effective fail-safe measure against inadvertently exposing symbols that should be hidden from clients.

The problem with this approach is that it hides the internal _set_name function from other modules in the library. If the library's developer trusts that any internal call to _set_name doesn't need to be validated but wants to validate all client calls, the symbol must be visible to other modules within the library but not to the library's client. Therefore, the static storage class is not appropriate to hide symbols from the client but disclose them to all the library's modules.

A second option for exposing only the symbols intended for client use is to have an exported symbols file that lists the symbols to export; all other symbols are hidden. Listing 3 shows the export_list file.

**Listing 3**  File listing the names of the symbols to export

```
# File: export_list
_name
_set_name
```

To compile the library, you use the gcc -exported_symbols_list option to specify the file containing the names of the symbols to export, as shown here:

```
gcc -dynamiclib Person.c -exported_symbols_list export_list -o libPerson.dylib
```

The third and most convenient option for exposing only name and set_name is to set the visibility attribute in their implementations to "default" and set the gcc -fvisibility option to hidden when compiling the library's source files. Listing 4 shows how the Person.c file looks after setting the visibility attribute for the symbols to be exported.

**Listing 4**  Person module using visibility attribute to export symbols

```
/* File: Person.c */
#include "Person.h"
#include <string.h>

// Symbolic name for visibility("default") attribute.
#define EXPORT __attribute__((visibility("default")))

static char _person_name[30] = {'\0'};

EXPORT                        // Symbol to export
char* name(void) {
    return _person_name;
}

void _set_name(char* name) {
   strcpy(_person_name, name);
}

EXPORT                        // Symbol to export
void set_name(char* name) {
    if (name == NULL) {
        _set_name("");
    }
```

```
    else {
        _set_name(name);
    }
}
```

The library would then be compiled using the following command:

```
% gcc -dynamiclib Person.c -fvisibility=hidden -o libPerson.dylib
```

The `gcc -fvisibility=hidden` option tells the compiler to set the visibility of any symbols without a visibility attribute to hidden, thereby hiding them from the library's clients. For details on the `visibility` attribute and the `gcc -fvisibility` option, see *GNU C/C++/Objective-C 4.0.1 Compiler User Guide* and the `gcc` man page.

Following these symbol-exporting guidelines ensures that libraries export only the symbols you want to make available to your clients, simplifying the use of the library by its clients and facilitating its maintenance by its developers. The document *How to Write Shared Libraries* provides an in-depth analysis of symbol exporting strategies. This document is available at http://people.redhat.com/drepper/dsohowto.pdf.

# Locating External Resources

When you need to locate resources your library needs at runtime—such as frameworks, images, and so on—you can use either of the following methods:

■  **Executable-relative location.** To specify a file path relative to the location of the main executable, not the referencing library, place the `@executable_path` macro at the beginning of the path. For example, in an application package that contains private frameworks (which, in turn, contain shared libraries), any of the libraries can locate an application resource called `MyImage.tiff` inside the package by specifying the path `@executable_path/../Resources/MyImage.tiff`. Because `@executable_path` resolves to the binary inside the `MacOS` directory in the application bundle, the resource file path must specify the `Resources` directory as a subdirectory of the `MacOS` parent directory (the `Contents` directory). For a detailed discussion of directory bundles, see *Bundle Programming Guide*.

■  **Library-relative location.** To specify a file path relative to the location of the library itself, place the `@loader_path` macro at the beginning of the pathname. Library-relative location allows you to locate library resources within a directory hierarchy regardless of where the main executable is located.

# Library Dependencies

When you develop a dynamic library, you specify its dependent libraries by linking your source code against them. When a client of your library tries to load it, your library's dependent libraries must be present in the file system for your library to load successfully. Depending on how the client loads your library, some or all of your library's references to symbols exported by its dependent libraries are resolved. You should consider using the `dlsym` function to get the address of symbols when they are needed instead of having references that may always have to be resolved at load time. See "Using Symbols" (page 39) for details.

The more dependent libraries your library has, the longer it takes for your library to load. Therefore, you should link your library against only those dynamic libraries required at load time. After you compile your library, you can view its dependent libraries in a shell editor with the `otool -L` command-line tool.

Any dynamic libraries your library seldom uses or whose functionality is needed only when performing specific tasks should be used as runtime loaded libraries; that is, they should be opened with the `dlopen` function. For example, when a module in your library needs to perform a task that requires the use of a nondependent library, the module should use `dlopen` to load the library, use the library to perform its task, and close the library with `dlclose` when finished. For additional information on loading libraries at runtime, see "Opening Dynamic Libraries" (page 35).

You should also keep to a minimum the number of external references to symbols in dependent libraries. This practice optimizes further your library's load time.

You must disclose to your library's users all the libraries your library uses and whether they are dependent libraries. When users of your dynamic library link their images, the static linker must be able to find all your library's dependent libraries, either through the link line or symbolic links. Also, because your dynamic library loads successfully even when some or all the libraries it opens at runtime are not present at load time, users of your library must know which dynamic libraries your library opens at runtime and under which circumstances. Your library's users can use that information when investigating unexpected behavior by your library.

# Module Initializers and Finalizers

When dynamic libraries are loaded, they may need to prepare resources or perform special initialization before doing anything else. Conversely, when the libraries are unloaded, they may need to perform some finalization processes. These tasks are performed by **initializer functions** and **finalizer functions**, also called constructors and destructors, respectively.

> **Note:** Applications can also define and use initializer and finalizers. However, this section focuses on their use in dynamic libraries.

Initializers can safely use symbols from dependent libraries because the dynamic loader executes the static initializers of an image's dependent libraries before invoking the image's static initializers.

You indicate that a function is an initializer by adding the `constructor` attribute to its definition. The `destructor` attribute identifies finalizer functions. Initializers and finalizers must not be exported. A dynamic library's initializers are executed in the order they are encountered by the compiler. It's finalizers, on the other hand, are executed in the reverse order as encountered by the compiler.

For example, Listing 5 shows a set of initializers and finalizers defined identically in two files `File1.c` and `File2.c` in a dynamic library called Inifi.

**Listing 5**        Inifi initializer and finalizer definitions

```
/* Files: File1.c, File2.c */
#include <stdio.h>
__attribute__((constructor))
static void initializer() {
    printf("[%s] initializer()\n", __FILE__);
```

```
}

__attribute__((constructor))
static void initializer2() {
    printf("[%s] initializer2()\n", __FILE__);
}

__attribute__((constructor))
static void initializer3() {
    printf("[%s] initializer3()\n", __FILE__);
}

__attribute__((destructor))
static void finalizer() {
    printf("[%s] finalizer()\n", __FILE__);
}

__attribute__((destructor))
static void finalizer2() {
    printf("[%s] finalizer2()\n", __FILE__);
}

__attribute__((destructor))
static void finalizer3() {
    printf("[%s] finalizer3()\n", __FILE__);
}
```

Continuing the example, the Inifi dynamic library is the sole dependent library of the Trial application, generated from the `Trial.c` file, shown in Listing 6.

**Listing 6**      The Trial.c file

```
/* Trial.c */
#include <stdio.h>
int main(int argc, char** argv) {
    printf("[%s] Finished loading. Now quitting.\n", __FILE__);
    return 0;
}
```

Listing 7 shows the output produced by the Trial application.

**Listing 7**      Execution order of a dynamic library's initializers and finalizers

```
% gcc -dynamiclib File1.c File2.c -fvisibility=hidden -o libInifi.dylib
% gcc Trial.c libInifi.dylib -o trial
% ./trial
[File1.c] initializer1()
[File1.c] initializer2()
[File1.c] initializer3()
[File2.c] initializer1()
[File2.c] initializer2()
[File2.c] initializer3()
[Trial.c] Finished loading. Now quitting.
[File2.c] finalizer3()
[File2.c] finalizer2()
[File2.c] finalizer1()
[File1.c] finalizer3()
[File1.c] finalizer2()
```

```
[File1.c] finalizer1()
```

Although you can have as many static initializers and finalizers in an image as you want, you should consolidate your initialization and finalization code into one initializer and one finalizer per module, as needed. You may also choose to have one initializer and one finalizer per library.

In Mac OS X v10.4 and later, static initializers can access the arguments given to the current application. By defining the initializer's parameters as you would define the parameters to an application's main function, you can get the number of arguments given, the arguments themselves, and the process's environment variables. In addition, to guard against an initializer or finalizer being called twice, you should conditionalize your initialization and finalization code inside the function. Listing 8 shows the definition of a static initializer that has access to the application's arguments and conditionalizes its initialization code.

**Listing 8**        Definition of a static initializer

```
__attribute__((constructor))
static void initializer(int argc, char** argv, char** envp) {
static initialized = 0;
if (!initialized) {
// Initialization code.
initialized = 1;
}
}
```

**Note:** Some systems support a naming convention for initializers and finalizers, `_init` and `_fini`, respectively. This convention is not supported in Mac OS X v10.4.

# C++–Based Libraries

Using C++ to implement a dynamic library presents a couple of challenges—mainly exporting symbol names and creating and destroying objects. The following sections detail how to export symbols from a C++–based dynamic library and how to provide clients with functions that create and destroy instances of a class.

## Exporting C++ Symbols

C++ uses **name mangling** to encode size and type information in a symbol name. Name mangling in C++ makes exporting symbols in a standard way across different platforms impossible. When a dynamic library is compiled, each symbol is renamed to include that information, but the encoding used is not standard between platforms. The dynamic loader uses string matching to locate symbols at runtime; the name of the symbol searched must match exactly the name of the target. When the symbol names have been mangled, the dynamic loader has no way of knowing how the name of the symbol it's searching for has been encoded.

To export nonmember symbols from a C++–based dynamic library so that the dynamic loader can find it, you must add the `extern "C"` directive to the symbols' declarations. This keyword tells the compiler not to mangle the symbol name. For example, the following declaration makes the `NewPerson` function available to the library's clients:

```
extern "C" Person* NewPerson(void);
```

Without this directive, the function name could be changed to `_Z9NewPersonv`, which would make it impossible for the dynamic loader to find the `NewPerson` symbol at runtime.

The only nonmember functions that must be exported are constructors and destructors, especially in dynamic libraries that can be used by clients as dependent libraries instead of runtime-loaded libraries. This is because clients must have access to a class's constructors and destructors so that they can use the `new` and `delete` operators on the class.

# Defining C++ Class Interfaces

A dynamic library should always publish its public interface to clients through header files. (Although clients can use dynamic libraries without their header files, doing so is very difficult and is prone to error.) The header file for a class that's available to clients must include the declarations of its public methods. Dynamic libraries that make a class available to its clients must include the virtual keyword in the declaration of all the class's methods, except for its constructors and destructors. For example, Listing 9 shows the declaration for the Person class.

**Listing 9**        Declaration for the Person class

```
class Person {
    private:
        char _person_name[30];
    public:
        Person();
        virtual void set_name(char person_name[]);
        virtual char* name();
};
```

# Creating and Destroying C++ Objects

When a C++–based dynamic library is a dependent library of its client, the client can use the `new` and `delete` operators to create and destroy instances of a class the library defines. However, clients that open a C++–based library at runtime through `dlopen` do not have access to the library's constructors because the constructors are exported with their names mangled, preventing the dynamic loader from locating them. See Listing 11 (page 29) for details about name mangling.

Clients that load a library at runtime to use a class must have a way to create and destroy a class's objects without using the `new` and `delete` operators. A library provides this functionality to its clients by exporting at least two class factory functions. **Class factory functions** create and destroy objects of a class on behalf of a class's user. Creator functions create an object of the class and return a pointer to it. Destructor functions dispose of an object created by a creator function for the same class. In addition to the factory functions, the library must define a data type for each exported factory function.

For example, Listing 10 shows the header and implementation files of the Person class, exported by the Person library. The header file includes the declaration and type definition of a pair of factory functions, `NewPerson`, and `DeletePerson`:

**Listing 10**        Interface and implementation of a C++ class in a dynamic library

```
/* File: Person.h */
```

```
class Person {
    private:
        char _person_name[30];
    public:
        Person();
        virtual void set_name(char person_name[]);
        virtual char* name();
};

// Constructor function and function type.
extern "C" Person* NewPerson(void);
typedef Person * Person_creator(void);

// Destructor function and function type.
extern "C" void DeletePerson(Person* person);
typedef void Person_disposer(Person*);

/* File: Person.cpp */
#include <iostream>
#include "Person.h"

#define EXPORT __attribute__((visibility("default")))

EXPORT
Person::Person() {
    this->set_name("<no value>");
}

EXPORT
Person* NewPerson(void) {
    return new Person;
}

EXPORT
void DeletePerson(Person* person) {
    delete person;
}

void Person::set_name(char name[]) {
    strcpy(_person_name, name);
}

char* Person::name(void) {
    return _person_name;
}
```

Listing 11 shows how a client might use the Person library.

**Listing 11**    Client using a C++ class implemented in a runtime loaded library

```
/* File: Client.cpp */

#include <iostream>
#include <dlfcn.h>
#include "Person.h"

int main() {
    using std::cout;
```

**29**

```
    using std::cerr;

    // Open the library.
    void* lib_handle = dlopen("./libPerson.dylib", RTLD_LOCAL);
    if (!lib_handle) {
        cerr << "[" << __FILE__ << "] main: Unable to open library: "
            << dlerror() << "\n";
        exit(EXIT_FAILURE);
    }

    // Get the NewPerson function.
    Person_creator* NewPerson = (Person_creator*)dlsym(lib_handle, "NewPerson");
    if (!NewPerson) {
        cerr << "[" << __FILE__ << "] main: Unable to find NewPerson method: "
            << dlerror() << "\n";
        exit(EXIT_FAILURE);
    }

    // Get the DeletePerson function.
    Person_disposer* DeletePerson =
        (Person_disposer*)dlsym(lib_handle, "DeletePerson");
    if (!DeletePerson) {
        cerr << "[" << __FILE__
            << "] main: Unable to find DeletePerson method: "
            << dlerror() << "\n";
        exit(EXIT_FAILURE);
    }

    // Create Person object.
    Person* person = (Person*)NewPerson();

    // Use Person object.
    cout << "[" << __FILE__ << "] person->name() = " << person->name() << "\n";
    person->set_name("Floriane");
    cout << "[" << __FILE__ << "] person->name() = " << person->name() << "\n";

    // Destroy Person object.
    DeletePerson(person);

    // Close the library.
    if (dlclose(lib_handle) != 0) {
        cerr << "[" << __FILE__ << "] main: Unable to close library: "
            << dlerror() << "\n";
    }

    return 0;
}
```

For further details on how clients use C++ classes implemented in dynamic libraries, see "Using C++ Classes" (page 43).

# Objective-C–Based Libraries

There are a few issues to consider while designing or updating an Objective-C–based dynamic library:

- Publishing the public interface of an Objective-C class or category is different from the way symbols are exported in C.

  In Objective-C every method of every class is available at runtime. Clients can introspect classes to find out which methods are available. However, so that client developers don't receive a flurry of warnings about missing method implementations, library developers should publish the interface to their classes and categories as protocols to client developers.

- Objective-C–based libraries have access to more initialization facilities than those available to C-based libraries.

- Objective-C has an class-alias facility that allows library developers to rename classes in a revision but allow clients to link against that revision to continue using the names used in earlier revisions.

The following sections explore these areas in detail.

## Defining Class and Category Interfaces

Because client developers generally don't have access to the implementation of Objective-C classes and categories defined in dynamic libraries, library developers must publish the public interfaces of classes and categories as protocols in header files. Client developers compile their products using those header files and are able to instantiate the classes correctly by adding the necessary protocol names to variable definitions. Listing 12 shows the header and implementation files of the Person class in an Objective-C–based library. Listing 13 shows the header and implementation files of the Titling category in the same library, which adds the -setTitle method to the Person class.

**Listing 12**     Header and implementation files of the Person class

```
/* File: Person.h */
#import <Foundation/Foundation.h>

@protocol Person
- (void)setName:(NSString*)name;
- (NSString*)name;
@end

@interface Person : NSObject <Person> {
    @private
NSString* _person_name;
}
@end

/* File: Person.m */
#import <Foundation/Foundation.h>
#import "Person.h"

@implementation Person
- (id)init {
    if (self = [super init]) {
        _person_name = @"";
    }
}

- (void)dealloc {
    [_person_name release];
```

```
    [super dealloc];
}

- (void)setName:(NSString*)name {
    [_person_name release];
    _person_name = [name retain];
}

- (NSString*)name {
    return _person_name;
}
@end
```

**Listing 13**      Header and implementation files of the Titling category to the Person class

```
/* File: Titling.h */
#import <Foundation/Foundation.h>
#import "Person.h"

@protocol Titling
- (void)setTitle:(NSString*)title;
@end

@interface Person (Titling) <Titling>
@end

/* File: Titling.m */
#import <Foundation/Foundation.h>
#import "Titling.h"

@implementation Person (Titling)
- (void)setTitle:(NSString*)title {
    [self setName:[[title stringByAppendingString:@" "]
        stringByAppendingString:[self name]]];
}
@end
```

Listing 14 shows how a client might use the library.

**Listing 14**      Client using the Person library

```
/* File: Client.m */
#import <Foundation/Foundation.h>
#import <dlfcn.h>
#import "Person.h"
#import "Titling.h"

int main() {
    NSAutoreleasePool* pool = [NSAutoreleasePool new];

    // Open the library.
    void* lib_handle = dlopen("./libPerson.dylib", RTLD_LOCAL);
    if (!lib_handle) {
        NSLog(@"[%s] main: Unable to open library: %s\n",
            __FILE__, dlerror());
        exit(EXIT_FAILURE);
    }
```

```
    // Get the Person class (required with runtime-loaded libraries).
    Class Person_class = objc_getClass("Person");
    if (!Person_class) {
        NSLog(@"[%s] main: Unable to get Person class", __FILE__);
        exit(EXIT_FAILURE);
    }

    // Create an instance of Person.
    NSLog(@"[%s] main: Instantiating Person_class", __FILE__);
    NSObject<Person,Titling>* person = [Person_class new];

    // Use person.
    [person setName:@"Perrine LeVan"];
    [person setTitle:@"Ms."];
    NSLog(@"[%s] main: [person name] = %@", __FILE__, [person name]);

    // Dispose of person.
    [person release];

    // Close the library.
    if (dlclose(lib_handle) != 0) {
        NSLog(@"[%s] Unable to close library: %s\n",
            __FILE__, dlerror());
        exit(EXIT_FAILURE);
    }

    [pool release];
    return(EXIT_SUCCESS);
}
```

## Initializing Objective-C Classes

Objecive-C–based dynamic libraries provide several initialization facilities for modules, classes, and categories. The following list describes those facilities in the order they are executed.

1. The +load method: Initializes resources needed by a class or a category. The Objective-C runtime sends the load message to every class a library implements; it then sends the load message to every category the library implements. The order in which sibling classes are sent the load message is undetermined. Implement the +load method to initialize resources needed by a class or category. Note that there's no corresponding "unload" method.

2. Module initializers: Initializes a module. The dynamic loader calls all the initializer functions (defined with the constructor attribute) in each of a library's modules. See "Module Initializers and Finalizers" (page 25) for more information on module initializers.

3. The +initialize method: Initializes resources needed by instances of a class before any instances are created. The Objective-C runtime sends the initialize message to a class just before creating an instance of the class. Note that there's no corresponding "finalize" message sent to a class when the library is unloaded or the process terminates.

## Creating Aliases to a Class

When you rename a class in a revision of a dynamic library, you can reduce the adoption burden to client developers by adding an alias to the new name in the library's header file. This practice allows client developers to release clients that take advantage of the new version of the library quickly. Client developers can later update references to the class at their leisure.

# Design Guidelines Checklist

This list provides a summary of the guidelines for improving specific aspects of a dynamic library:

- Ease of use
  - ❏ Reduce the number of symbols a library exports.
  - ❏ Provide unique names to public interfaces.

- Ease of maintenance
  - ❏ Export accessor functions to variables. Don't export variables.
  - ❏ Implement public interfaces as wrappers to internal, private interfaces.

- Performance
  - ❏ Minimize the number of references to symbols in dependent libraries. Use `dlsym(RTLD_GLOBAL, <symbol_name>)` to obtain the address of symbols exported by dependent libraries when they are needed.
  - ❏ Minimize the number of dependent libraries. Consider loading libraries with `dlopen` when absolutely necessary. Remember to close the library with `dlclose` when done.
  - ❏ Implement public interfaces as wrappers to internal, private interfaces.

- Compatibility
  - ❏ Export symbols as weakly linked symbols.
  - ❏ Encode a library's major version number in its filename.

# Dynamic Library Usage Guidelines

The dynamic loader compatibility functions provide a portable and efficient way to load code at runtime. However, using the functions incorrectly can degrade application performance. This article shows how to correctly load and use dynamic libraries in your applications.

Dynamic libraries help to distribute an application's functionality into distinct modules that can be loaded as they are needed. Dynamic libraries can be loaded either when the application launches or as it runs. Libraries that are loaded at launch time are called **dependent libraries**. Libraries that are loaded at runtime are called **dynamically loaded libraries**. You specify which dynamic libraries your application depends on by linking your application against them. However, it's more efficient to use dynamic libraries as dynamically loaded libraries instead of dependent libraries. That is, you should open libraries when you're about to use symbols they export and close them when you're done. In some cases, the system unloads dynamically loaded libraries when it determines that they aren't being used.

This article uses the word **image** to refer to an application file or a dynamic library. Application files contain the application's code and the code from the static libraries the application uses. The dynamic libraries the application loads at launch time or runtime are separate images.

## Opening Dynamic Libraries

The dynamic loader loads an image's dependent libraries when the image is opened; that is, when an application is loaded or when an dynamic library is opened. The dynamic loader binds references to symbols exported by dependent libraries lazily. Lazy binding means the symbol references are bound only when the image actually uses the symbols. As a debugging measure, you can specify that all references to the exported symbols of a library be bound when the dynamic loader opens the library. You use the `gcc -bind_at_load` flag when generating the dynamic library.

To use a dynamic library that is not a dependent library of your image, you use the `dlopen` function. This function tells the dynamic loader to load a specific dynamic library into the address space of the current process. This function also allows you to specify when the dynamic loader binds the library's references to the corresponding exported symbols in its dependent libraries and whether to place the library's exported symbols in the current process's global scope or a local scope. This function returns a handle called **library handle**. This handle represents the dynamically loaded library in calls to `dlsym` (to use an exported symbol) and `dlclose` (to close the library). The library handle provides `dlsym` a limited domain within which to search for a symbol (see "Using Symbols" (page 39) for details). The client must call `dlclose` when it's finished using the dynamically loaded library (for example, when the module that opened the library has finished its task).

A dynamic library may itself have dependent libraries. To find out which libraries a dynamic library depends on, use the `otool -L` command. Before using the library, you must ensure that all its dependent libraries are present in your computer. Otherwise, the dynamic loader doesn't load your application or library when requested at application launch time or when the library is opened with `dlopen`.

A process can open the same dynamic library several times without closing it. The `dlopen` function returns the same library handle it returned in the first call, but it also increments the reference count associated with the handle. Calls to `dlclose` decrement the library handle's reference count. Therefore, you must balance every call to `dlopen` with a call to `dlclose`. When the reference count for a library handle reaches 0, the dynamic loader may remove the library from the address space of the application.

## The Library Search Process

The first parameter to `dlopen` is the name of the dynamic library to open. This may be a filename or a partially or fully qualified pathname. For example, `libCelsus.dylib`, `lib/libCelsus.dylib`, or `/usr/local/libCelsus.dylib`.

The dynamic loader searches for libraries in the directories specified by a set of environment variables and the process's current working directory. These variables, when defined, must contain a colon-separated list of pathnames (absolute or relative) in which the dynamic loader searches for libraries. Table 1 lists the variables.

**Table 1**    Environment variables that define dynamic-loader search paths

| Environment variable | Default value |
|---|---|
| LD_LIBRARY_PATH | No default value |
| DYLD_LIBRARY_PATH | No default value |
| DYLD_FALLBACK_LIBRARY_PATH | $HOME/lib;/usr/local/lib;/usr/lib |

When the library name is a filename (that is, when it doesn't include directory names), the dynamic loader searches for the library in several locations until it finds it, in the following order:

1. `$LD_LIBRARY_PATH`

2. `$DYLD_LIBRARY_PATH`

3. The process's working directory

4. `$DYLD_FALLBACK_LIBRARY_PATH`

When the library name contains at least one directory name, that is, when the name is a pathname (relative or fully qualified), the dynamic loader searches for the library in the following order:

1. `$DYLD_LIBRARY_PATH` using the filename

2. The given pathname

3. `$DYLD_FALLBACK_LIBRARY_PATH` using the filename

For example, say you set the environment variables introduced earlier as shown in the following table.

| Environment variable | Value |
|---|---|
| LD_LIBRARY_PATH | ./lib |
| DYLD_LIBRARY_PATH | /usr/local/dylibs |
| DYLD_FALLBACK_LIBRARY_PATH | /usr/local/lib |

Assuming your application calls dlopen with the filename libCelsus.dylib, the dynamic loader would attempt to open the library using the following pathnames, in order:

| Pathname | Description |
|---|---|
| ./lib/libCelsus.dylib | LD_LIBRARY_PATH environment variable |
| /usr/local/dylibs/libCelsus.dylib | DYLD_LIBRARY_PATH environment variable |
| libCelsus.dylib | Current working directory |
| /usr/local/lib/libCelsus.dylib | DYLD_FALLBACK_LIBRARY_PATH environment variable |

If the application calls dlopen with the pathname /libs/libCelsus.dylib, the dynamic loader tries to find the library using these pathnames, in order:

| Pathname | Description |
|---|---|
| /usr/local/dylibs/libCelsus.dylib | DYLD_LIBRARY_PATH environment variable |
| /libs/libCelsus.dylib | Path as given |
| /usr/local/lib/libCelsus.dylib | DYLD_FALLBACK_LIBRARY_PATH environment variable |

## Specifying the Scope and Binding Behavior of Exported Symbols

The second parameter of the dlopen function specifies two properties: the scope of the library's exported symbols in the current process and when to bind the application's references the those symbols.

Symbol scope directly affects the performance of applications. Therefore, it's important that you set the appropriate scope for a library you open at runtime.

A dynamically loaded library's exported symbols can be in one of two levels of scope in the current process: global and local. The main difference between the scopes is that the symbols in the global scope are available to all images in the process, including other dynamically loaded libraries. Symbols in the local scope can be used only by the image that opened the library. See "Using Symbols" (page 39) for more information.

When the dynamic loader searches for symbols, it performs string comparisons against every symbol in the search scope. Reducing the number of symbols the dynamic loader has to go through to find the desired symbol improves your application's performance. Opening all dynamically loaded libraries into the local scope instead of the global scope maximizes symbol search performance.

You should never need to open a dynamic library into the process's global scope so that all modules in the application have access to its symbols. Instead, each module that uses the library should open it into its local scope. When done, the module should close the library. If you want the symbols exported by the library to be available to all images in the process, consider making the library a dependent library of the application.

The parameter used to specify symbol scope is also used to specify when the undefined external symbols of the dynamically loaded library are resolved (or bound with their definitions the library's own dependent libraries). Undefined external symbols of dynamically loaded libraries can be resolved either immediately or lazily. If a client application uses immediate binding when opening a dynamic library with `dlopen`, the dynamic loader binds all the undefined external symbols of the dynamically loaded library before returning control to the client application. For example, Listing 1 shows the log messages the dynamic loader produces when the `DYLD_PRINT_BINDINGS` environment variable is set and a client application loads a dynamic library called `libArt.dylib`:

**Listing 1**      Bindings resolved during call to `dlopen` using immediate binding

```
dyld: bind: client:_dlopen$lazy_ptr = libSystem.B.dylib:_dlopen
dyld: bind: libArt.dylib:_dlopen$lazy_ptr = libSystem.B.dylib:_dlopen
dyld: bind: libArt.dylib:_printf$LDBLStub$lazy_ptr = libA.dylib:_printf$LDBLStub
dyld: bind: libArt.dylib:_dlerror$lazy_ptr = libSystem.B.dylib:_dlerror
dyld: bind: libArt.dylib:_dlclose$lazy_ptr = libSystem.B.dylib:_dlclose
dyld: bind: libArt.dylib:_dlsym$lazy_ptr = libSystem.B.dylib:_dlsym
dyld: bind: libArt.dylib:___stub_getrealaddr$lazy_ptr = libA.dylib:___stub_getrealaddr
dyld: bind: libArt.dylib:_NSIsSymbolNameDefinedWithHint$lazy_ptr =
                    libSystem.B.dylib:_NSIsSymbolNameDefinedWithHint
dyld: bind: libArt.dylib:_strcpy$lazy_ptr = libSystem.B.dylib:_strcpy
dyld: bind: libArt.dylib:_NSLookupAndBindSymbolWithHint$lazy_ptr =
                    libSystem.B.dylib:_NSLookupAndBindSymbolWithHint
dyld: bind: libArt.dylib:_NSAddressOfSymbol$lazy_ptr =
                    libSystem.B.dylib:_NSAddressOfSymbol
dyld: bind: libArt.dylib:_strlen$lazy_ptr = libSystem.B.dylib:_strlen
```

The first log message indicates that the client application's _dlopen undefined symbol was bound. The remaining messages are the bindings the dynamic loader performs on the dynamic library as part of the loading process before returning control to the calling routine. When using lazy binding, the dynamic loader resolves only the client application's reference to the `dlopen` function, returning control to the calling routine much sooner. For more information on dynamic loader logging, see "Logging Dynamic Loader Events" (page 79).

Once a library has been opened with `dlopen`, the scope defined for it cannot be changed by subsequent calls to `dlopen` to load the same library. For example, if the process opens a library that hasn't been loaded into the local scope and later opens the same library into the global scope, the opened library retains its local status. That is, the symbols the library exports do not become available in the global scope with the latter call. This is true even if the library is closed before reopening it within the same process.

> **Important:** All runtime loaded should be opened into the local scope. Adhering to this rule makes finding symbols at runtime as fast as possible.

Immediate binding slows the loading of dynamic libraries, especially when those libraries contain many undefined external symbols. However, immediate binding can help during development and testing of dynamic libraries because when the dynamic loader cannot resolve all the undefined external symbols of a dynamically loaded library, the application terminates with an error. When deploying the application, however, you should use lazy loading because undefined external symbols are bound only when necessary. Loading dynamic libraries this way can help make your application feel more responsive to its users.

The external undefined symbols in dependent libraries are bound when they are first used unless the client image's compile line includes the `gcc -bind_at_load` option. See the `ld` man page for details.

# Using Symbols

After opening a dynamic library using `dlopen`, an image uses the `dlsym` function to get the address of the desired symbol before using it. This function takes two parameters. The first one specifies in which libraries the dynamic loader looks for the symbol. The second parameter specifies the name of the symbol. For example:

```
symbol_pointer = dlsym(library_handle, "my_symbol")
```

This invocation tells the dynamic loader to search for a symbol named `my_symbol` among the symbols exported by the dynamically loaded library represented by the `library_handle` variable.

There are three scopes the dynamic loader can search for a symbol: a specific dynamic library, the current image's dependent libraries, and the global scope of the process:

■ **The local scope:** To search the symbols exported by a particular dynamic library that has been loaded using `dlopen`, you provide `dlsym` with the handle to that library. This is the most efficient usage model.

■ **The next scope:** This search scope is useful only when a module has interposed a symbol exported by a dependent library. For example, you may need to intercept all calls to a system function to perform bookkeeping before calling the real implementation. In that case, in your custom definition of the function, you get the address of the function you interposed by invoking `dlsym` with the `RTLD_NEXT` special handle instead of the handle to a particular library. Such a call returns the address of the function that would have been executed if you hadn't masked out that implementation with your own. Therefore, only the dependent libraries of the current image are searched; any other libraries, including libraries opened by the image making the `dlsym` call, are not searched. Also, in a flat namespace, the search starts in the first dependent library listed after the current one when the application was linked.

■ **The global scope:** To search the global scope, you call `dlsym` with the `RTLD_DEFAULT` special handle. The dynamic loader searches the dependent libraries (loaded at launch time) and the dynamically loaded libraries (loaded at runtime with `RTLD_GLOBAL`) for the first match of the symbol name given to `dlsym`. You should avoid performing global symbol searches because they are the most inefficient.

> **Note:** Name conflicts between dynamic shared libraries are not discovered at compile time, link time, or runtime. The `dlsym` function uses string matching to find symbols. If two libraries use the same name for a function, the first one that matches the symbol name given to `dlsym` is returned.

To illustrate the concepts introduced in this section, take the application depicted in Figure 1. It shows that the application has two dependent libraries, `libArt.dylib` and `libBus.dylib`. The `libBus.dylib` library itself has two dependent libraries, `libBus1.dylib` and `libBus2.dylib`. The `libBus1.dylib` library has one dependent library, `libBus1a.dylib`. In addition, there are four dynamic libraries the application doesn't depend on, `libCar.dylib`, `libCar1.dylib`, `libDot.dylib`, and `libDot1.dylib`. The `libCar1.dylib` library is a dependent library of `libCar.dylib` and `libDot1.dylib` is a dependent library of `libDot.dylib`. All the libraries except `libArt.dylib` export the `dependencies` function. Each library has a unique implementation of the `...name` function.

**Figure 1**      Application with dependent library hierarchy
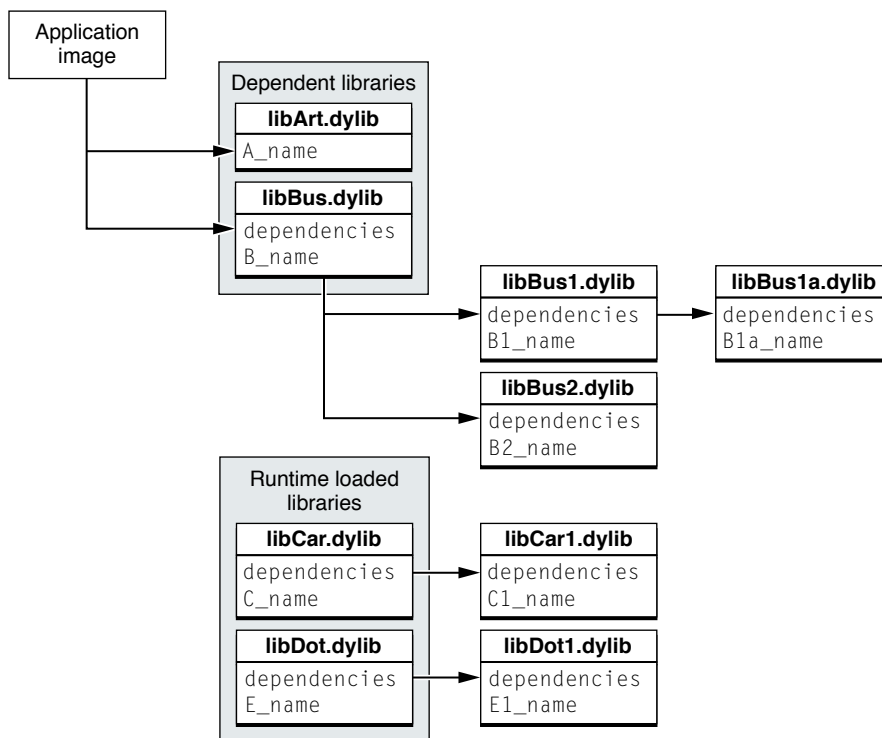


The application image can access the exported symbols in `libArt.dylib` and `libBus.dylib` directly, as shown in Listing 2.

**Listing 2**      Application image using symbols exported by dependent libraries through undefined external references

```
#include <stdio.h>

extern char* A_name();          // libArt.dylib
extern char* dependencies();    // libBus.dylib

int main(void) {
    printf("[%s] libArt.A_name() = %s\n", __FILE__, A_name());
```

```
    printf("[%s] libBus.dependencies() = %s\n", __FILE__, dependencies());
}
```

The application image, however, cannot directly access the symbols exported by `libBus1.dylib`, `libBus1a.dylib`, and `libBus2.dylib` because those libraries are not dependent libraries of the application image. To gain access to those symbols, the application image has to open the corresponding libraries using `dlopen`, as shown in Listing 3.

**Listing 3**      Application image using a symbol exported by a dynamic library loaded at runtime

```
#include <stdio.h>
#include <dlfcn.h>

int main(void) {
    void* Bus1a_handle = dlopen("libBus1a.dylib", RTLD_LOCAL);
    if (Bus1a_handle) {
        char* (*b1a_name)() = dlsym(Bus1a_handle, "B1a_name");
        if (b1a_name) {
            printf("[%s] libBus1a.B1a_name() = %s\n",
                __FILE__, b1a_name());
        }
    }
    else {
        printf("[%s] Unable to open libBus1a.dylib: %s\n",
            __FILE__, dlerror());
    }
    dlclose(Bus1a_handle);
}
```

So far you have seen how to access symbols either through references to imported symbols or by obtaining the address of the desired symbol by calling `dlsym` with the handle of the corresponding library or with the `RTLD_DEFAULT` special handle. As mentioned earlier, interposed symbols offer the ability to change the definition of a symbol exported by a dependent library.

To access the original definition of interposed symbols, you call `dlsym` with the `RTLD_NEXT` special handle. Listing 4 shows the implementation of the `dependencies` function in the Bus library (the implementation is identical in Bus1 and Bus1a). The function in Bus returns the name of the library (contained in the `k_lib_name` variable) concatenated with a separator string and the text returned by the next definition of `dependencies`, which is found in the Bus1 library. The definition in Bus1 concatenates its name with a separator string and the text returned by the definition in Bus1a. The definition in Bus1a is the last that would've been found if none of the client images had defined their own version. Therefore, when Bus1a invokes `dlsym(RTLD_NEXT, "dependencies")` no other definitions for `dependencies` are found. That's the end of the interposition hierarchy of the `dependencies` function.

**Listing 4**      Library image using an interposed symbol

```
#include <string.h>
static char* k_lib_name = "libBus";
char* dependencies(void) {
    char _dependencies[50] = "";
    strcpy(_dependencies, k_lib_name);
    char* (*next_dependencies)() =
        dlsym(RTLD_NEXT, "dependencies");// look for next definition
    if (next_dependencies) {
```

```
        strncat(_dependencies, ", ",
            sizeof(_dependencies) - strlen(_dependencies) - 1);
        strncat(_dependencies, next_dependencies(),
            sizeof(_dependencies) - strlen(_dependencies) - 1);
    }
    return strdup(_dependencies);
}
```

When the application image calls the dependencies function in the Bus library, it obtains the names of all the libraries the Bus library depends on, as shown in Listing 5.

**Listing 5**      Application image calling an interposed function

```
#include <stdio.h>
extern char* dependencies();     // libBus.dylib

int main(void) {
    printf("[%s] libBus.dependencies() = %s\n",
        __FILE__, dependencies());
}
```

# Using Weakly Linked Symbols

To promote compatibility with earlier or later revisions, a dynamic library may export some or all its public symbols as weakly linked symbols. A **weakly linked symbol** is one for which the compiler generates a weak reference when a client is compiled against a library. Weakly linked symbols may have the `weak_import` attribute in their declarations in the library's header files, or the library's developer may otherwise document which of the library's public symbols are weakly linked. A third way to identify weakly linked symbols it by executing the command:

```
nm -m <client_file> | grep weak
```

This command lists the weakly linked symbols imported from dependent libraries.

A weakly linked symbol may or may not be defined by a dependent library. That is, although the symbol is declared in a header file, the corresponding dynamic library file may not contain an implementation of that symbol. Listing 6 shows how a weakly linked symbol may be declared in a header file for a dynamic library. Clients that use this header file as their interface to the corresponding dependent library are guaranteed that `name` and `set_name` are defined. However, `clear_name` may not be implemented. The dependent library loads successfully whether or not it implements `clear_name`. But it doesn't load if it doesn't define either `name` or `set_name` . When the library doesn't implement a weakly linked symbol, the dynamic loader sets to 0 any client references to the symbol.

**Listing 6**      Header file with a weakly linked symbol declaration

```
/* File: Person.h */
#define WEAK_IMPORT __attribute__((weak_import))

char* name(void);

void set_name(char* name);

WEAK_IMPORT
```

```
void clear_name(void);
```

Weakly linked symbols are used by library developers to maximize the compatibility of a client with earlier or newer versions of a dependent library. For example, a symbol that was implemented in a particular revision of a library may not be available in a later revision. But a client linked against the first revision also works with the second revision. Client developers, however, must ensure the existence of the symbol in the running process before executing it. This mechanism is also used to provide a standard interface to plug-ins, which may or may not implement the entire interface.

Listing 7 shows code that ensures that a particular function is defined before using it. When the function is not found, the client uses a different function to accomplish the desired task. In this case, the fallback function is not a weakly linked symbol, so no test is required. Other situations may not offer an alternate interface. In such cases the client may not be able to perform the desired task.

**Listing 7**     Using a weakly linked symbol

```
// Clear the 'name' property.
if (clear_name) {
    clear_name();
}
else {
    set_name(" ");
}
```

# Using C++ Classes

How client developers use a C++ class depends on whether the dynamic library that implements it is loaded when the client is loaded (dependent library) or at a later point (runtime loaded library). Dependent-library classes can be used directly. That is, clients can create and delete objects with the `new` and `delete` operators. Classes implemented in libraries loaded at runtime with `dlopen` are called **runtime loaded classes**.

A runtime loaded class must be instantiated by the client using that class's factory functions, declared as part of the class's interface. **Factory functions** create and destroy instances of a specific class: Constructor functions instantiate objects and destructor functions destroy them. Clients must use factory functions instead of `new` and `delete` because the dynamic loader doesn't have access to a runtime loaded class's constructors and destructors. When the client calls a factory function, the library invokes the appropriate constructor and destructor on the client's behalf. After you create an instance of a runtime loaded class, you invoke its member functions the same way you would call them if the class were defined locally.

The interface for C++ classes implemented in dynamic libraries is made up of at least the class declaration and a set of factory functions. The class interface includes one type definition per constructor function. To use a factory function, you must create an object of the appropriate type and get the address of the function with `dlsym`. You can then call the factory function to create or destroy an object of the class.

Listing 8 shows the interface to the Person class, implemented in the Person library.

**Listing 8**     C++ class interface

```
/* File: Person.h */
```

```
class Person {
    private:
        char _person_name[30];
    public:
        Person();
        Person(char* name);
        virtual void set_name(char person_name[]);
        virtual char* name();
};

// Constructor functions and function types.
extern "C" Person* NewPerson(void);
typedef Person * Person_creator(void);
extern "C" Person* NewPersonWithName(char name[]);
typedef Person * PersonWithName_creator(char name);

// Destructor function and function type.
extern "C" void DeletePerson(Person* person);
typedef void Person_disposer(Person*);
```

Note that the Person class has two constructor functions, `NewPerson` and `NewPersonWithName`. Each function declaration has a corresponding type, `Person_creator` and `PersonWithName_creator`. Listing 9 shows how a client may use the Person library.

**Listing 9**    Client using a C++ runtime loaded class

```
/* File: Client.cpp */
#include <iostream>
#include <dlfcn.h>
#include "Person.h"

int main() {
    using std::cout;
    using std::cerr;

    // Open the library.
    void* lib_handle = dlopen("./libPerson.dylib", RTLD_LOCAL);
    if (!lib_handle) {
        exit(EXIT_FAILURE);
    }

    // Get the NewPerson function.
    Person_creator* NewPerson = (Person_creator*)dlsym(lib_handle, "NewPerson");
    if (!NewPerson) {
        exit(EXIT_FAILURE);
    }

    // Get the NewPersonWithName function.
    PersonWithName_creator* NewPersonWithName =
(PersonWithName_creator*)dlsym(lib_handle, "NewPersonWithName");
    if (!NewPersonWithName) {
        exit(EXIT_FAILURE);
    }

    // Get the DeletePerson function.
    Person_disposer* DeletePerson =
        (Person_disposer*)dlsym(lib_handle, "DeletePerson");
    if (!DeletePerson) {
```

```
        exit(EXIT_FAILURE);
    }

    // Create Person objects.
    Person* person1 = NewPerson();
    Person* person2 = NewPersonWithName("Cendrine");

    // Use Person objects.
    person1->set_name("Floriane");
    cout << "[" << __FILE__ << "] person1->name() = " << person1->name() <<
"\n";
    person2->set_name("Marcelle");
    cout << "[" << __FILE__ << "] person2->name() = " << person2->name() <<
"\n";

    // Destroy Person objects.
    DeletePerson(person1);
    DeletePerson(person2);

    // Close the library.
    if (dlclose(lib_handle) != 0) {
        exit(EXIT_FAILURE);
    }

    return 0;
}
```

# Using Objective-C Classes

To use an Objective-C class or category implemented in a dynamic library, a client should have an interface to the class or category. With knowledge of the class's correct interface, the client can create instances of the class that are appropriately typed. Otherwise, the compiler produces warnings for methods with missing declarations.

The interfaces of Objective-C classes and categories are published in the library's header files as protocols. Instantiating a class implemented in a dependent library is no different from doing the same for a locally defined class. However, when you load a dynamic library at runtime using dlopen, you must obtain the appropriate class by calling the objc_getClass function.

For example, Listing 10 contains the interfaces for the Person class and the Titling category to that class, which are implemented by the Person dynamic library.

**Listing 10**     Interface to the Person class and its Titling category

```
/* File: Person.h */
#import <Foundation/Foundation.h>

@protocol Person
- (void)setName:(NSString*)name;
- (NSString*)name;
@end

@interface Person : NSObject <Person> {
    @private
```

```
NSString* _person_name;
}
@end

/* File: Titling.h */
#import <Foundation/Foundation.h>
#import "Person.h"

@protocol Titling
- (void)setTitle:(NSString*)title;
@end

@interface Person (Titling) <Titling>
@end
```

A client compiled with these interfaces and linked against the Person library can create objects that implement the interfaces in a very straightforward way, as shown in Listing 11.

**Listing 11**      Example of a client that uses the Person library as a dependent library

```
/* File: Client.m */
#import <Foundation/Foundation.h>
#import "Person.h"
#import "Titling.h"

int main() {
    NSAutoreleasePool* pool = [NSAutoreleasePool new];

    // Create an instance of Person.
    Person<Titling>* person = [Person new];

    // Use person.
    [person setName:@"Perrine"];
    NSLog(@"[%s] main: [person name] = %@", __FILE__, [person name]);
    [person setTitle:@"Ms."];
    NSLog(@"[%s] main: [person name] = %@", __FILE__, [person name]);

    // Dispose of the Person instance.
    [person release];

    [pool release];
    return EXIT_SUCCESS;
}
```

When the Person library is a runtime loaded library, however, the client must obtain a reference to the Person class from the Objective-C runtime after loading the library, using `objc_getClass`. It can then use that reference to instantiate a Person object. However, the variable that holds the instance must by typed as an NSObject that implements the Person and Titling protocols to avoid compiler warnings. When done, the client closes the library, as shown in "Using Weakly Linked Symbols."

**Listing 12**      Example of a client that uses the Person library as a runtime loaded library

```
/* File: Client.m */
#import <Foundation/Foundation.h>
#import <dlfcn.h>
#import "Person.h"
#import "Titling.h"
```

```
int main() {
    NSAutoreleasePool* pool = [NSAutoreleasePool new];

    // Open the Person library.
    void* lib_handle = dlopen("./libPerson.dylib", RTLD_LOCAL);
    if (!lib_handle) {
        exit(EXIT_FAILURE);
    }

    // Get the Person class (required with runtime loaded libraries).
    Class Person_class = objc_getClass("Person");
    if (!Person_class) {
        exit(EXIT_FAILURE);
    }

    // Create an instance of Person.
    NSObject<Person,Titling>* person = [Person_class new];

    // Use person.
    [person setName:@"Perrine LeVan"];
    NSLog(@"[%s] main: [person name] = %@", __FILE__, [person name]);
    [person setTitle:@"Ms."];
    NSLog(@"[%s] main: [person name] = %@", __FILE__, [person name]);

    // Dispose of person.
    [person release];

    // Close the Person library.
    if (dlclose(lib_handle) != 0) {
        exit(EXIT_FAILURE);
    }

    [pool release];
    return EXIT_SUCCESS;
}
```

# Getting Information on an Address

One of the dynamic loader compatibility (DLC) functions, `dladdr`, provides information on the image and nearest symbol that corresponds to an address. You can use this function to obtain information about the library that exports a particular symbol.

The information `dladdr` provides is returned through an output parameter of type `Dl_info`. These are the names of the structure's fields as well as their descriptions:

■  `dli_fname`: The pathname of the image

■  `dli_fbase`: The base address of the image within the process

■  `dli_sname`: The name of the symbol with an address that is equal to or lower than the address provided to `dladdr`

■  `dli_saddr`: The address of the symbol indicated by `dli_sname`

Listing 13 shows how an application image can get information about a symbol:

**Listing 13**    Getting information about a symbol

```
#include <stdio.h>
#include <dlfcn.h>

extern char* dependencies();

int main(void) {
    // Get information on dependencies().
    Dl_info info;
    if (dladdr(dependencies, &info)) {
        printf("[%s] Info on dependencies():\n", __FILE__);
        printf("[%s]    Pathname: %s\n",          __FILE__, info.dli_fname);
        printf("[%s]    Base address: %p\n",      __FILE__, info.dli_fbase);
        printf("[%s]    Nearest symbol: %s\n",    __FILE__, info.dli_sname);
        printf("[%s]    Symbol address: %p\n",    __FILE__, info.dli_saddr);
    }
    else {
        printf("[%s] Unable to find image containing the address %x\n",
    __FILE__, &dependencies);
    }
}
```

# Creating Dynamic Libraries

When you create or update dynamic libraries, it's important to consider the library's ease of use by developers who incorporate them in their products. It's also important to give these developers flexibility by allowing their products to work with earlier or later versions of the libraries they used while developing their products.

This article demonstrates how to write a dynamic library so that it's easy to use by developers who want to take advantage of it in their own development. This article also describes how to update existing libraries and manage their version information to maximize client compatibility.

## Creating Libraries

When creating a dynamic library, you should perform these tasks:

- Define the library's purpose: This information provides the focus required to define the library's public interface.

- Define the library's interface (header files): This is the interface through which the library's clients access its functionality.

- Implement the library (implementation files): This is where you define the public functions that the library's clients use. You may also need to define private variables and functions needed to implement the interface but not required by clients.

- Set the library's version information: The library's version information is divided in three parts: major, minor, and compatibility. You specify all the parts of the library's version information when you create the `.dylib` file. See "Managing Client Compatibility With Dependent Libraries" (page 16) for details.

- Test the library: At the very least, you should define a test for each of the public functions your library exposes, to ensure that they perform the correct action given particular test inputs or after performing a specific set of operations.

The following sections provide an example of the process taken to develop a simple dynamic library, called Ratings. The files mentioned in the following sections are included in `Ratings/1.0` in this document's companion-file package.

## Defining the Library's Purpose

The purpose of the Ratings library is to provide a ratings analyzer to its clients. Ratings are strings made up of asterisks (*) that represent levels of satisfaction for a particular item. For example, in Apple's iTunes application, you can specify whether you like a particular song a lot with a five-star rating (*****) or not at all with a one-star rating (*).

The initial release of the library provides a way for clients to add ratings, get a count of the ratings they have added, get the mean rating, and clear the rating set.

## Defining the Library's Interface

Before you implement the library, you must define the interface the library's clients use to interact with it. You should carefully specify each function's semantics. A clear definition benefits you because it makes clear the purpose of each public function. It also benefits developers who use your library because it tells them exactly what to expect when they call each function in their code.

This is an example of a list of interface functions with each function's semantic meaning:

- `void addRating(char* rating)`: Adds a rating value to the set kept by the library. The rating argument is a string; it must not be NULL. Each character in the string represents one rating point. For example, an empty string (`""`) is equivalent to a rating of 0, `"*"` means , `"***"` means 3, and so on. The actual characters used in the string are immaterial. Therefore, `"  "` means 1 and `"3456"` means 4. Each call to this function increments the value returned by the ratings function.
- `int ratings(void)`: Returns the number of rating values in the set. This function has no side effects.
- `char* meanRatings(void)`: Returns the mean rating in the rating set as a string, using one asterisk per rating point. This function has no side effects.
- `clearRatings(void)`: Clears the rating set. After calling this function (with no subsequent calls to `addRating`), the `ratings` function returns `0`.

Listing 1 (page 50) shows the header file that the Ratings library's clients include to access its interface.

**Listing 1**       Interface to Ratings 1.0

```
/* File: Ratings.h
 * Interface to libRatings.A.dylib 1.0.
 ***********************************/

/* Adds 'rating' to the set.
 *      rating: Each character adds 1 to the numeric rating
 *              Example: "" = 0, "*" = 1, "**" = 2, "wer " = 4.
 */
void addRating(char* rating);

/* Returns the number of ratings in the set.
 */
int ratings(void);

/* Returns the mean rating of the set.
 */
```

```
char* meanRating(void);

/* Clears the set.
 */
void clearRatings(void);
```

# Implementing the Library

The interface declared in "Defining the Library's Interface" (page 50) is implemented in Ratings.c , shown in Listing 2 (page 51).

**Listing 2**      Implementation of Ratings 1.0

```
/* File: Ratings.c
 * Compile with -fvisibility=hidden.                // 1
 ********************************/

#include "Ratings.h"
#include <stdio.h>
#include <string.h>

#define EXPORT __attribute__((visibility("default")))
#define MAX_NUMBERS 99

static int _number_list[MAX_NUMBERS];
static int _numbers = 0;

// Initializer.
__attribute__((constructor))
static void initializer(void) {                     // 2
    printf("[%s] initializer()\n", __FILE__);
}

// Finalizer.
__attribute__((destructor))
static void finalizer(void) {                       // 3
    printf("[%s] finalizer()\n", __FILE__);
}

// Used by meanRating, middleRating, frequentRating.
static char* _char_rating(int rating) {
    char result[10] = "";
    int int_rating = rating;
    for (int i = 0; i < int_rating; i++) {
        strncat(result, "*", sizeof(result) - strlen(result) - 1);
    }
    return strdup(result);
}

// Used by addRating.
void _add(int number) {                             // 4
    if (_numbers < MAX_NUMBERS) {
        _number_list[_numbers++] = number;
    }
}
```

```
// Used by meanRating.
int _mean(void) {
    int result = 0;
    if (_numbers) {
        int sum = 0;
        int i;
        for (i = 0; i < _numbers; i++) {
            sum += _number_list[i];
        }
        result = sum / _numbers;
    }
    return result;
}

EXPORT
void addRating(char* rating) {                                // 5
    if (rating != NULL) {
        int numeric_rating = 0;
        int post = 0;
        while (*rating++ != '\0' && pos++ < 5) {
            numeric_rating++;
        }
        _add(numeric_rating);
    }
}

EXPORT
char* meanRating(void) {
    return _char_rating(_mean());
}

EXPORT
int ratings(void) {
    return _numbers;
}

EXPORT
void clearRatings(void) {
    _numbers = 0;
}
```

The following list describes the tagged lines.

1.  This comment is only to remind the library's developers to compile this file with the `gcc -fvisibility=hidden` option, so that only the symbols with the `visibility("default")` attribute are exported.

2.  This initializer is defined only to show at which point of a client's execution the library's initializers are called by the dynamic loader.

3.  This finalizer is defined only to show at which point of client's execution the library's finalizers are called by the dynamic loader.

4.  The `_add` function is an example of an internal function. Clients don't need to know about it and, therefore, it's not exported. Also, because internal calls are trusted, no validation is performed. However, there's no requirement that internal-use functions lack validation.

5.  The `addRating` function is an example of an exported function. To ensure that only correct ratings are added, the function's input parameter is validated.

# Setting the Library's Version Information

When you compile the library's source files into a `.dylib` file, you set version information that specifies whether clients can use versions of the library earlier or later than the version they were linked against. When the client is loaded into a process, the dynamic loader looks for the `.dylib` file in the library search paths and, if it finds it, compares the version information of the `.dylib` file with the version information recorded in the client image. If the client is not compatible with the `.dylib` file, the dynamic loader doesn't load the client into the process. In effect, the client's load process is aborted because the dynamic loader was unable to locate a compatible dependent library.

Listing 3 (page 53) shows the command used to generate version 1.0 of the Ratings library.

**Listing 3**      Generating version 1.0 of the Ratings dynamic library

```
[Ratings/1.0]% make dylib
gcc -dynamiclib -std=gnu99 Ratings.c -current_version 1.0 -compatibility_version
 1.0 -fvisibility=hidden -o libRatings.A.dylib
```

This list indicates where the major version, minor version, and compatibility version are specified:

■   The major version number is specified in the library's filename as "A" in `-o libRatings.A.dylib`.

■   The minor version number is specified in `-current_version 1.0`.

■   The compatibility version number is specified in `-compatibility_version 1.0`.

> **Note:** You can use `libtool` to create a dynamic library from a set of object files.

# Testing the Library

Before publishing a dynamic library, you should test its public interface to ensure that it performs as you specified in the interface's documentation (see "Defining the Library's Interface" (page 50). To provide maximum flexibility to clients, you should make sure that your library can be used as a dependent library (clients link against it, and the library is loaded when the client is loaded) or as a runtime loaded library (clients don't link against it and use dlopen to load it).

Listing 4 (page 53) shows an example of a test client that uses the Ratings library as a dependent library.

**Listing 4**      Testing Ratings 1.0 as a dependent library

```
/* Dependent.c
 * Tests libRatings.A.dylib 1.0 as a dependent library.
 ****************************************************/

#include <stdio.h>
#include <string.h>
#include "Ratings.h"
```

```
#define PASSFAIL "Passed":"Failed"
#define UNTST "Untested"

int main(int argc, char** argv) {
    printf("[start_test]\n");

    // Setup.
    addRating(NULL);
    addRating("");
    addRating("*");
    addRating("**");
    addRating("***");
    addRating("*****");
    addRating("*****");

    // ratings.
    printf("[%s] ratings(): %s\n",
        __FILE__, (ratings() == 6? PASSFAIL));

    // meanRating.
    printf("[%s] meanRating(): %s\n",
        __FILE__, (strcmp(meanRating(), "**") == 0)? PASSFAIL);

    // clearRatings.
    clearRatings();
    printf("[%s] clearRatings(): %s\n",
        __FILE__, (ratings() == 0? PASSFAIL));

    printf("[end_test]\n");
    return 0;
}
```

The following command generates the Dependent client application. Note that libRatings.A.dylib is included in the compile line.

```
gcc Dependent.c libRatings.A.dylib -o Dependent
```

The Dependent application produces output that shows whether calling each of the library's exported functions produces the expected results. In addition, the initializer and finalizer functions defined in the library produce output lines that indicate when they are invoked in relation to the application's normal process. This output is shown in Listing 5 (page 54).

**Listing 5**    Test results for Ratings 1.0 as a dependent library

```
> ./Dependent
[Ratings.c] initializer()
[start_test]
[Dependent.c] ratings(): Passed
[Dependent.c] meanRating(): Passed
[Dependent.c] clearRatings(): Passed
[end_test]
[Ratings.c] finalizer()
```

Notice that initializer is called before the main function. The finalizer function, on the other hand, is called after exiting the main function.

Testing the Ratings library as a runtime loaded library requires another test application, Runtime. Listing 6 (page 55) shows it's source file.

**Listing 6**       Testing Ratings 1.0 as a runtime loaded library

```
/* Runtime.c
 * Tests libRatings.A.dylib 1.0 as a runtime loaded library.
 **********************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <dlfcn.h>
#include <string.h>
#include "Ratings.h"

#define PASSFAIL "Passed":"Failed"
#define UNTST "Untested"

int main(int argc, char** argv) {
    printf("[start_test]\n");

    // Open the library.
    char* lib_name = "./libRatings.A.dylib";
    void* lib_handle = dlopen(lib_name, RTLD_NOW);
    if (lib_handle) {
        printf("[%s] dlopen(\"%s\", RTLD_NOW): Successful\n", __FILE__, lib_name);
    }
    else {
        printf("[%s] Unable to open library: %s\n",
            __FILE__, dlerror());
        exit(EXIT_FAILURE);
    }

    // Get the symbol addresses.
    void (*addRating)(char*) = dlsym(lib_handle, "addRating");
    if (addRating) {
        printf("[%s] dlsym(lib_handle, \"addRating\"): Successful\n", __FILE__);
    }
    else {
        printf("[%s] Unable to get symbol: %s\n",
            __FILE__, dlerror());
        exit(EXIT_FAILURE);
    }
    char* (*meanRating)(void) = dlsym(lib_handle, "meanRating");
    if (meanRating) {
        printf("[%s] dlsym(lib_handle, \"meanRating\"): Successful\n", __FILE__);
    }
    else {
        printf("[%s] Unable to get symbol: %s\n",
            __FILE__, dlerror());
        exit(EXIT_FAILURE);
    }
    void (*clearRatings)(void) = dlsym(lib_handle, "clearRatings");
    if (clearRatings) {
        printf("[%s] dlsym(lib_handle, \"clearRatings\"): Successful\n", __FILE__);
    }
    else {
        printf("[%s] Unable to get symbol: %s\n",
```

```
            __FILE__, dlerror());
        exit(EXIT_FAILURE);
    }
    int (*ratings)(void) = dlsym(lib_handle, "ratings");
    if (ratings) {
        printf("[%s] dlsym(lib_handle, \"ratings\"): Successful\n", __FILE__);
    }
    else {
        printf("[%s] Unable to get symbol: %s\n",
            __FILE__, dlerror());
        exit(EXIT_FAILURE);
    }

    // Setup.
    addRating(NULL);
    addRating("");
    addRating("*");
    addRating("**");
    addRating("***");
    addRating("*****");
    addRating("*****");

    // ratings.
    printf("[%s] ratings(): %s\n", __FILE__, (ratings() == 6? PASSFAIL));

    // meanRating.
    printf("[%s] meanRating(): %s\n", __FILE__, (strcmp(meanRating(), "**") == 0)?
PASSFAIL);

    // clearRatings.
    clearRatings();
    printf("[%s] clearRatings(): %s\n", __FILE__, (ratings() == 0? PASSFAIL));

    // Close the library.
    if (dlclose(lib_handle) == 0) {
        printf("[%s] dlclose(lib_handle): Successful\n", __FILE__);
    }
    else {
        printf("[%s] Unable to open close: %s\n",
            __FILE__, dlerror());
    }

    printf("[end_test]\n");
    return 0;
}
```

The Runtime application is very similar to the Dependent application. However, Runtime must load libRuntime.A.dylib using the dlopen function. After that, it must get the address of each function exported by the library using dlsym before using it.

The following command generates the Runtime client application.

```
[Ratings/1.0]% make runtime
gcc Runtime.c -o Runtime
```

Listing 7 (page 57) shows the output produced by Runtime.

**Listing 7**    Test results for Ratings 1.0 as a runtime loaded library

```
> ./Runtime
[start_test]
[Ratings.c] initializer()
[Runtime.c] dlopen("./libRatings.A.dylib", RTLD_NOW): Successful
[Runtime.c] dlsym(lib_handle, "addRating"): Successful
[Runtime.c] dlsym(lib_handle, "meanRating"): Successful
[Runtime.c] dlsym(lib_handle, "clearRatings"): Successful
[Runtime.c] dlsym(lib_handle, "ratings"): Successful
[Runtime.c] ratings(): Passed
[Runtime.c] meanRating(): Passed
[Runtime.c] clearRatings(): Passed
[Runtime.c] dlclose(lib_handle): Successful
[end_test]
[Ratings.c] finalizer()
```

Note that the Ratings library's initializer function is called within the main function execution before the call to `dlopen` returns, which differs from its execution point in the Dependent application Listing 5 (page 54). The `finalizer` function, however, is called after `main` has exited, the same point at which it's called in Dependent's execution. You should consider this when writing dynamic library initializers and finalizers.

# Updating Libraries

Making revisions to a previously published dynamic library is a delicate task. If you want existing clients to be able to use the library (that is, load the new version of the `.dylib` file without recompilation), you must ensure that the API those clients know about is unchanged, including its semantic meaning.

When you can guarantee that the API of the new version of the library is compatible with the API that clients linked against earlier versions know about, you can deem the new version a minor revision. When you want to publish a minor revision of a dynamic library to users of the library's clients (for example, the users of an application that uses your library) the only version information item you need to change is the library's current version. The major version (filename) and compatibility version of the library must remain the same. When end users replace the early version of the library with the new version in their computers, the library's clients use the new version without any problems. "Managing Client Compatibility With Dependent Libraries" (page 16).

## Making Compatible Changes

From the point of view of a client image, there are two sides to compatibility with the dynamic libraries it uses: compatibility with versions of a library later than the one the client knows about, known as forward compatibility, and compatibility with versions of a library earlier than the one the client is familiar with, known as backward compatibility. You maintain forward compatibility by ensuring that the library's API remains compatible across revisions. You facilitate backward compatibility by exporting new functions as weak references. In turn, your library's clients must ensure that weakly imported functions actually exist before using them.

The Ratings library, introduced in "Creating Libraries" (page 49), has a second version, 1.1. Listing 8 (page 58) shows the updated header for the Ratings library.

**Listing 8**    Interface to Ratings 1.1

```
/* File: Ratings.h
 * Interface to libRatings.A.dylib 1.1.
 ************************************/

#define WEAK_IMPORT __attribute__((weak_import))

/* Adds 'rating' to the set.
 *     rating: Each character adds 1 to the numeric rating
 *     Example: "" = 0, "*" = 1, "**" = 2, "wer " = 4.
 */
void addRating(char* rating);

/* Returns the number of ratings in the set.
 */
int ratings(void);

/* Returns the mean rating of the set.
 */
char* meanRating(void);

/* Returns the medianRating of the set.
 */
WEAK_IMPORT
char* medianRating(void);                       // 1

/* Returns the most frequent rating of the set.
 */
WEAK_IMPORT
char* frequentRating(void);                     // 2

/* Clears the set.
 */
void clearRatings(void);
```

The tagged lines declare the new functions. Notice that both declarations include the weak_import attribute, informing client developers that the function is weakly linked. Therefore, clients must make sure the function exists before calling it.

Listing 9 (page 58) shows the library's new implementation file.

**Listing 9**    Implementation of Ratings 1.1

```
/* Runtime.c
 * Tests libRatings.A.dylib 1.1 as a runtime loaded library.
 ********************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <dlfcn.h>
#include <string.h>
#include "Ratings.h"

#define PASSFAIL "Passed":"Failed"
#define UNTST "Untested"

int main(int argc, char** argv) {
```

```
    printf("[start_test]\n");

    // Open the library.
    char* lib_name = "./libRatings.A.dylib";
    void* lib_handle = dlopen(lib_name, RTLD_NOW);
    if (lib_handle) {
        printf("[%s] dlopen(\"%s\", RTLD_NOW): Successful\n", __FILE__, lib_name);
    }
    else {
        printf("[%s] Unable to open library: %s\n",
            __FILE__, dlerror());
        exit(EXIT_FAILURE);
    }

    // Get the function addresses.
    void (*addRating)(char*) = dlsym(lib_handle, "addRating");
    if (addRating) {
        printf("[%s] dlsym(lib_handle, \"addRating\"): Successful\n", __FILE__);
    }
    else {
        printf("[%s] Unable to get symbol: %s\n",
            __FILE__, dlerror());
        exit(EXIT_FAILURE);
    }
    char* (*meanRating)(void) = dlsym(lib_handle, "meanRating");
    if (meanRating) {
        printf("[%s] dlsym(lib_handle, \"meanRating\"): Successful\n", __FILE__);
    }
    else {
        printf("[%s] Unable to get symbol: %s\n",
            __FILE__, dlerror());
        exit(EXIT_FAILURE);
    }
    void (*clearRatings)(void) = dlsym(lib_handle, "clearRatings");
    if (clearRatings) {
        printf("[%s] dlsym(lib_handle, \"clearRatings\"): Successful\n", __FILE__);
    }
    else {
        printf("[%s] Unable to get symbol: %s\n",
            __FILE__, dlerror());
        exit(EXIT_FAILURE);
    }
    int (*ratings)(void) = dlsym(lib_handle, "ratings");
    if (ratings) {
        printf("[%s] dlsym(lib_handle, \"ratings\"): Successful\n", __FILE__);
    }
    else {
        printf("[%s] Unable to get symbol: %s\n",
            __FILE__, dlerror());
        exit(EXIT_FAILURE);
    }
    char* (*medianRating)(void) = dlsym(lib_handle, "medianRating");       // weak
import
    char* (*frequentRating)(void) = dlsym(lib_handle, "frequentRating");    // weak
import

    // Setup.
    addRating(NULL);
```

```
    addRating("");
    addRating("*");
    addRating("**");
    addRating("***");
    addRating("*****");
    addRating("*****");

    // ratings.
    printf("[%s] ratings(): %s\n", __FILE__, (ratings() == 6? PASSFAIL));

    // meanRating.
    printf("[%s] meanRating(): %s\n", __FILE__, (strcmp(meanRating(), "**") == 0)?
PASSFAIL);

    // medianRating.
    if (medianRating) {
        printf("[%s] medianRating(): %s\n", __FILE__, (strcmp(medianRating(), "**") ==
 0? PASSFAIL));
    }
    else {
        printf("[%s] medianRating(): %s\n", __FILE__, UNTST);
    }

    // frequentRating.
    if (frequentRating) {
        char* mfr = "*****";
        printf("[%s] frequentRating(): %s\n", __FILE__, strncmp(mfr, frequentRating(),
 sizeof(mfr)) == 0? PASSFAIL);
    }
    else {
        printf("[%s] mostFrequentRating(): %s\n", __FILE__, UNTST);
    }

    // clearRatings.
    clearRatings();
    printf("[%s] clearRatings(): %s\n", __FILE__, (ratings() == 0? PASSFAIL));

    // Close the library.
    if (dlclose(lib_handle) == 0) {
        printf("[%s] dlclose(lib_handle): Successful\n", __FILE__);
    }
    else {
        printf("[%s] Unable to open close: %s\n",
            __FILE__, dlerror());
    }

    printf("[end_test]\n");
    return 0;
}
```

## Updating the Library's Version Information

shows the command used to generate version 1.1 of the Ratings library.

**Listing 10**     Generating version 1.1 of the Ratings dynamic library

```
[Ratings/1.1]% make dylib
gcc -dynamiclib -std=gnu99 Ratings.c /usr/local/lib/libAverages.dylib -current_version
 1.1 -compatibility_version 1.0 -fvisibility=hidden -o libRatings.A.dylib
```

Notice that this time the library's current version is set to 1.1 but its compatibility version remains 1.0. When a client is linked against version 1.1 of this library, the compatibility version is encoded in the client image. Therefore, the dynamic loader loads the client image whether it finds version 1.1 or 1.0 of libRatings.A.dylib. That is, the client is backwards compatible with version 1.0 of the library. For details on why /usr/local/lib/libAverages.dylib was added to the compile line, see "Using Dependent Libraries" (page 68).

> **Note:** Weak references are a feature of Mac OS X v10.2 and later. Therefore, when using weak references, your library can be used only by applications running in Mac OS X v10.2 or later.

## Testing the New Version of the Library

Similar to creating initial versions of a library, updating libraries require thorough testing. You should at least add tests for each function you added to your test applications. If you used weak references, your test applications should ensure the existence of the new functions before calling them.

The Ratings/1.1 directory in this document's companion-file package contains the source files for the Ratings 1.1 library. Listing 11 (page 61) shows an updated version of the Dependent application. The tagged lines show how to check for the presence of a function before using it.

**Listing 11**     Testing Ratings 1.1 as a dependent library

```
/* Dependent.c
 * Tests libRatings.A.dylib 1.1 as a dependent library.
 ****************************************************/

#include <stdio.h>
#include <string.h>
#include "Ratings.h"

#define PASSFAIL "Passed":"Failed"
#define UNTST "Untested"

int main(int argc, char** argv) {
    printf("[start_test]\n");

    // Setup.
    addRating(NULL);
    addRating("");
    addRating("*");
    addRating("**");
    addRating("***");
    addRating("*****");
    addRating("*****");

    // ratings.
    printf("[%s] ratings(): %s\n",
        __FILE__, (ratings() == 6? PASSFAIL));
```

```
        // meanRating.
        printf("[%s] meanRating(): %s\n",
            __FILE__, (strcmp(meanRating(), "**") == 0)? PASSFAIL);

        // medianRating.
        if (medianRating) {                            // 1
            printf("[%s] medianRating(): %s\n",
                __FILE__, (strcmp(medianRating(), "**") == 0? PASSFAIL));
        }
        else {
            printf("[%s] medianRating(): %s\n", __FILE__, UNTST);
        }

        // frequentRating.
        if (frequentRating) {                          // 2
            char* mfr = "*****";
            printf("[%s] frequentRating(): %s\n",
                __FILE__, strncmp(mfr, frequentRating(),
                sizeof(mfr)) == 0? PASSFAIL);
        }
        else {
            printf("[%s] mostFrequentRating(): %s\n",
                __FILE__, UNTST);
        }

        // clearRatings.
        clearRatings();
        printf("[%s] clearRatings(): %s\n",
            __FILE__, (ratings() == 0? PASSFAIL));

        printf("[end_test]\n");
        return 0;
    }
```

Ratings 1.1 depends on Averages 1.1. Therefore, to build the library as well as the test applications, libAverages.A.dylib must be installed in /usr/local. To accomplish this, run this command after opening this document's companion-file package:

```
[Avarages/1.0]% make install
```

To compile the new version of the library as well as the Dependent application, you must set the MACOSX_DEPLOYMENT_TARGET environment variable to 10.2 or later. These are commands needed to compile the library and the test applications from the Ratings/1.1 directory using tsch:

```
[Ratings/1.1]% setenv MACOSX_DEPLOYMENT_TARGET 10.4
[Ratings/1.1]% make
gcc -dynamiclib -std=gnu99 Ratings.c /usr/local/lib/libAverages.dylib -current_version
 1.1 -compatibility_version 1.0 -fvisibility=hidden -o libRatings.A.dylib
gcc Dependent.c libRatings.A.dylib /usr/local/lib/libAverages.dylib -o Dependent
gcc Runtime.c -o Runtime
```

The output the application produces is shown in Listing 12 (page 62).

**Listing 12**    Test results for Ratings 1.1 used as a dependent library

```
> ./Dependent
[start_test]
```

```
[Dependent.c] ratings(): Passed
[Dependent.c] meanRating(): Passed
[Dependent.c] medianRating(): Passed
[Dependent.c] frequentRating(): Passed
[Dependent.c] clearRatings(): Passed
[end_test]
```

Applications that use dlopen to load your library should not fail if they are unable to obtain an address for your weakly exported functions with dlsym. Therefore, the application that test your library as a runtime loaded library, should allow for the absence of weakly imported functions.

Listing 13 (page 63) shows an updated version of the Runtime application. Notice that it uses dlsym to try get the addresses of the new functions but doesn't exit with an error if they are unavailable. However, just before the application uses the new functions, it determines whether they actually exists. If they don't exist, it doesn't perform the test.

**Listing 13**      Testing Ratings 1.1 as a runtime loaded library

```c
/* Runtime.c
 * Tests libRatings.A.dylib 1.1 as a runtime loaded library.
 ********************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <dlfcn.h>
#include <string.h>
#include "Ratings.h"

#define PASSFAIL "Passed":"Failed"
#define UNTST "Untested"

int main(int argc, char** argv) {
    printf("[start_test]\n");

    // Open the library.
    char* lib_name = "./libRatings.A.dylib";
    void* lib_handle = dlopen(lib_name, RTLD_NOW);
    if (lib_handle) {
        printf("[%s] dlopen(\"%s\", RTLD_NOW): Successful\n", __FILE__, lib_name);
    }
    else {
        printf("[%s] Unable to open library: %s\n",
            __FILE__, dlerror());
        exit(EXIT_FAILURE);
    }

    // Get the function addresses.
    void (*addRating)(char*) = dlsym(lib_handle, "addRating");
    if (addRating) {
        printf("[%s] dlsym(lib_handle, \"addRating\"): Successful\n", __FILE__);
    }
    else {
        printf("[%s] Unable to get symbol: %s\n",
            __FILE__, dlerror());
        exit(EXIT_FAILURE);
    }
    char* (*meanRating)(void) = dlsym(lib_handle, "meanRating");
    if (meanRating) {
```

```
        printf("[%s] dlsym(lib_handle, \"meanRating\"): Successful\n", __FILE__);
    }
    else {
        printf("[%s] Unable to get symbol: %s\n",
            __FILE__, dlerror());
        exit(EXIT_FAILURE);
    }
    void (*clearRatings)(void) = dlsym(lib_handle, "clearRatings");
    if (clearRatings) {
        printf("[%s] dlsym(lib_handle, \"clearRatings\"): Successful\n", __FILE__);
    }
    else {
        printf("[%s] Unable to get symbol: %s\n",
            __FILE__, dlerror());
        exit(EXIT_FAILURE);
    }
    int (*ratings)(void) = dlsym(lib_handle, "ratings");
    if (ratings) {
        printf("[%s] dlsym(lib_handle, \"ratings\"): Successful\n", __FILE__);
    }
    else {
        printf("[%s] Unable to get symbol: %s\n",
            __FILE__, dlerror());
        exit(EXIT_FAILURE);
    }
    char* (*medianRating)(void) = dlsym(lib_handle, "medianRating");        // weak
import
    char* (*frequentRating)(void) = dlsym(lib_handle, "frequentRating");    // weak
import

    // Setup.
    addRating(NULL);
    addRating("");
    addRating("*");
    addRating("**");
    addRating("***");
    addRating("*****");
    addRating("*****");

    // ratings.
    printf("[%s] ratings(): %s\n", __FILE__, (ratings() == 6? PASSFAIL));

    // meanRating.
    printf("[%s] meanRating(): %s\n", __FILE__, (strcmp(meanRating(), "**") == 0)?
PASSFAIL);

    // medianRating.
    if (medianRating) {
        printf("[%s] medianRating(): %s\n", __FILE__, (strcmp(medianRating(), "**") ==
 0? PASSFAIL));
    }
    else {
        printf("[%s] medianRating(): %s\n", __FILE__, UNTST);
    }

    // frequentRating.
    if (frequentRating) {
        char* mfr = "*****";
```

```
        printf("[%s] frequentRating(): %s\n", __FILE__, strncmp(mfr, frequentRating(),
sizeof(mfr)) == 0? PASSFAIL);
    }
    else {
        printf("[%s] mostFrequentRating(): %s\n", __FILE__, UNTST);
    }

    // clearRatings.
    clearRatings();
    printf("[%s] clearRatings(): %s\n", __FILE__, (ratings() == 0? PASSFAIL));

    // Close the library.
    if (dlclose(lib_handle) == 0) {
        printf("[%s] dlclose(lib_handle): Successful\n", __FILE__);
    }
    else {
        printf("[%s] Unable to open close: %s\n",
            __FILE__, dlerror());
    }

    printf("[end_test]\n");
    return 0;
}
```

Listing 14 (page 65) shows the output produced by Runtime.

**Listing 14**      Test results for Ratings 1.1 used as a runtime loaded library

```
[Ratings/1.1]% ./Runtime
[start_test]
[Runtime.c] dlopen("./libRatings.A.dylib", RTLD_NOW): Successful
[Runtime.c] dlsym(lib_handle, "addRating"): Successful
[Runtime.c] dlsym(lib_handle, "meanRating"): Successful
[Runtime.c] dlsym(lib_handle, "clearRatings"): Successful
[Runtime.c] dlsym(lib_handle, "ratings"): Successful
[Runtime.c] ratings(): Passed
[Runtime.c] meanRating(): Passed
[Runtime.c] medianRating(): Passed
[Runtime.c] frequentRating(): Passed
[Runtime.c] clearRatings(): Passed
[Runtime.c] dlclose(lib_handle): Successful
[end_test]
```

To ensure that clients linked against version 1.0 of libRatings.A.dylib can use version 1.1 of the library, you can copy Ratings/1.1/libRatings.A.dylib to Ratings/1.0. When you run the first Dependent application, its output is the identical to the output it produced when it used version 1.0 of the library. The first Dependent application knows nothing about the functions introduces in version 1.1 of the Ratings library; therefore, it doesn't call them.

A more interesting test, however, is making sure that clients linked against version 1.1 of the Ratings library can run when their copy of the library is replaced with version 1.0. To test this, execute the following commands in Terminal:

```
[           ]% cd companion_dir/Ratings/1.1
[Ratings/1.1]% make
[Ratings/1.1]% cd ../1.0
[Ratings/1.0]% make
```

```
[Ratings/1.0]% cp libRatings.A.dylib ../1.1
[Ratings/1.0]% cd ../1.1
```

Listing 15 (page 66) shows the output produced by the second version Dependent, linked against Ratings 1.1, when it loads version 1.0 instead of version 1.1.

**Listing 15**    Test results for Ratings 1.0 used as a dependent library by a client linked against Ratings 1.1

```
[Ratings/1.1]% ./Dependent
[Ratings.c] initializer()
[start_test]
[Dependent.c] ratings(): Passed
[Dependent.c] meanRating(): Passed
[Dependent.c] medianRating(): Untested
[Dependent.c] mostFrequentRating(): Untested
[Dependent.c] clearRatings(): Passed
[end_test]
[Ratings.c] finalizer()
```

The second version of the Runtime test application produces similar output when using Ratings 1.0, as shown in Listing 16 (page 66).

**Listing 16**    Test results for Ratings 1.0 used as a runtime loaded library by a client developed against Ratings 1.1 when using Ratings 1.0

```
[Ratings/1.1]% ./Runtime
[start_test]
[Ratings.c] initializer()
[Runtime.c] dlopen("./libRatings.A.dylib", RTLD_NOW): Successful
[Runtime.c] dlsym(lib_handle, "addRating"): Successful
[Runtime.c] dlsym(lib_handle, "meanRating"): Successful
[Runtime.c] dlsym(lib_handle, "clearRatings"): Successful
[Runtime.c] dlsym(lib_handle, "ratings"): Successful
[Runtime.c] ratings(): Passed
[Runtime.c] meanRating(): Passed
[Runtime.c] medianRating(): Untested
[Runtime.c] mostFrequentRating(): Untested
[Runtime.c] clearRatings(): Passed
[Runtime.c] dlclose(lib_handle): Successful
[end_test]
[Ratings.c] finalizer()
```

# Using Dynamic Libraries

When you need to use a dynamic library in your product, you have to install the library in your computer. You may use dynamic libraries as dependent libraries (by specifying them in your product's link line) or as runtime loaded libraries (by loading them when they are needed, using `dlopen`).

This article describes the process of installing and using dynamic libraries. It's based on the Ratings dynamic library and the StarMeals, StarMeals2, and Grades applications, which are included in this document's companion-file package. This article also shows how to use dynamic libraries as dependent libraries or as runtime loaded libraries. Finally, this article demonstrates how to interpose the functions exported by a dynamic library.

## Installing Dependent Libraries

Before you can use a dynamic library as a dependent library, the library and its header files must be installed on your computer. The standard locations for header files are ~/include, /usr/local/include and `/usr/include`. The standard locations for dynamic libraries are `~/lib`, `/usr/local/lib`, and `/usr/lib`.

You may also place the .dylib fine at a nonstandard location in your filesystem, but you must add that location to one of these environment variables:

- `LD_LIBRARY_PATH`

- `DYLD_LIBRARY_PATH`

- `DYLD_FALLBACK_LIBRARY_PATH`

For details on how to add paths to these environment variables, see .

If you don't want to change the environment variables and you want to place the dynamic library in a nonstandard location, you must specify where in your filesystem you placed the library when you link your image. See the description of the gcc -dylib_file option in the gcc man page for details.

For example, in Mac OS X the executable code of applications can be packaged together with frameworks containing libraries created specifically for a particular application. These frameworks are known as **private embedded frameworks**. Applications that use private embedded frameworks,

as well as the frameworks themselves, must be specially built. See "Creating a Framework" in *Framework Programming Guide* and "Loading Code at Runtime" in *Mach-O Programming Topics* for details.

"Using Dependent Libraries" (page 68) requires that the Ratings 1.1 dynamic library be installed on your computer. To install Ratings 1.1 in your computer:

1.  Open this document's companion-file package.

2.  In Terminal, execute this command:

    ```
    [Ratings/1.1]% make install
    ```

> **Note:** To uninstall the Ratings dynamic library, execute the command `make uninstall`.

# Using Dependent Libraries

Using dynamic libraries as dependent libraries by linking your image against them provides several benefits, including producing smaller executable files and not having to get the address of the libraries' exported symbols before using them in your code. However, you still have to make sure a weakly imported symbol exists before using it.

All you need to do to use a dynamic library as a dependent library is include the library's headers in your source code and link the library against your application or library. The library's headers describe the symbols you can use. You should not use any other symbols to access the library's functionality. Otherwise, you may get unexpected results, or your image may stop working for its users when they update the dependent library in their computers.

Listing 1 (page 68) shows the source code of a small command-line based application that uses Ratings 1.1, developed in "Creating Dynamic Libraries" (page 49).

**Listing 1**      Using Ratings 1.1 as a dependent library

```
/* File: StarMeals.c
 * Uses functions in libRatings.A.dylib.
 *************************************/

#include <stdio.h>
#include <string.h>
#include <Ratings.h>

#define MAX_NAMES 100
#define MAX_NAME_LENGTH 30
#define MAX_RATING_LENGTH 5

static char* name_list[MAX_NAMES];
static char* rating_list[MAX_NAMES];
static int names = 0;

void addNameAndRating(char* name, char* rating) {
    name_list[names] = strdup(name);
```

```
    rating_list[names] = (strlen(rating) > MAX_RATING_LENGTH)? "*****" : strdup(rating);
    names++;
}

void test_data(void) {
    addNameAndRating("Spinach", "*");
    addNameAndRating("Cake", "****");
    addNameAndRating("Steak", "****");
    addNameAndRating("Caviar", "*");
    addNameAndRating("Broccoli", "****");
    addNameAndRating("Gagh", "*****");
    addNameAndRating("Chicken", "*****");
}

int main(int argc, char* argv[]) {
    int test_mode = 0;
    if (argc == 2) {
        if (strcmp(argv[1], "test") == 0) {
            test_mode = 1;
            printf("[start_test]\n");
            test_data();
        }
    }
    else {
        printf("Enter meal names and ratings in the form <name> <rating>.\n");
        printf("No spaces are allowed in the name and the rating.\n");
        printf("The rating can be * through *****.\n");
        printf("To finish, enter \"end\" for a meal name.\n");
        while (names < MAX_NAMES) {
            char name[MAX_NAME_LENGTH];
            char rating[MAX_RATING_LENGTH + 1];
            printf("\nName and rating: ");
            scanf("%s", &name);
            if (strcmp(name, "end") == 0) {
                break;
            }
            scanf("%s", rating);
            addNameAndRating(name, rating);
        }
        printf("\n");
    }

    if (names) {
        // Print data entered and call libRatings.addRating().
        printf("This is the data you entered:\n");
        for (int i = 0; i < names; i++) {
            printf("%s (%s)\n", name_list[i], rating_list[i]);
            addRating(rating_list[i]);
        }

        // Print statistical information.
        printf("\nThe mean rating is %s\n", meanRating());// 1
        if (medianRating) {      // 2
            printf("The median rating is %s\n", medianRating());
        }
        if (frequentRating) {    // 3
            printf("The most frequent rating is %s\n", frequentRating());
        }
```

```
        //printf("\n");
    }

    if (test_mode) {
        printf("[end_test]\n");
    }
    return 0;
}
```

This list describes the tagged lines:

■  Line 1: The meanRating function is guaranteed to exist in all versions of `libRating.A.dylib`. Therefore, no existence test is required.

■  Lines 2 and 3: The functions medianRating and frequentRating are available in Ratings 1.1 but not Ratings 1.0. Since StarMeals is to be backwards compatible with Ratings 1.0, it has to check for the existence of these functions before using them. Otherwise, StarMeals may crash.

> **Note:** Keep in mind that applications or dynamic libraries that use weak references, do not run on versions of Mac OS X earlier than 10.2.

To compile the `StarMeals.c` file, use the command shown in Listing 2 (page 70).

**Listing 2**      Compiling and linking StarMeals using only the link line

```
[StarMeals]% make
gcc -std=gnu99 StarMeals.c /usr/local/lib/libRatings.dylib
/usr/local/lib/libAverages.dylib -o StarMeals
```

Notice that the exact location of the library StarMeals directly depends (`libRatings.dylib`) is provided at the link line. The pathname `/usr/local/lib/libRatings.dylib` is actually a symbolic link to `/usr/local/lib/libRatings.A.dylib`. At link time, the static linker resolves the link and stores the library's actual filename in image it generates. With this approach, the dynamic linker always uses the library's complete name when it looks for an image's dependent libraries.

The static linker must also have access to the library the Ratings library depends on (`libAverages.A.dylib`), either through the link line (as Listing 2 (page 70) shows) or by having a symbolic link to that library in the current working directory, as shown in Listing 3 (page 70). This requirement stems from the fact that the static linker attempts to resolve all the dependent library's undefined external symbols. Any unresolved undefined external symbol causes the static linker to cancel the link operation. The location of libAverages.A.dylib at launch time doesn't necessarily have to be the same location of the library at link time.

**Listing 3**      Compiling and linking StarMeals using the link line and a symbolic link

```
[StarMeals]% ln -s /usr/local/lib/libAverages.A.dylib
[StarMeals]% gcc -std=gnu99 StarMeals.c /usr/local/lib/libRatings.A.dylib -o
StarMeals
```

Listing 4 (page 70) shows the output StarMeals produces when run in test mode:

**Listing 4**      Test output of the StarMeals application

```
> ./StarMeals test
```

```
Test mode. Data entered automatically.

This is the data you entered:
Spinach (*)
Cake (****)
Steak (****)
Caviar (*)
Broccoli (****)
Gagh (*****)
Chicken (*****)

The mean rating is ***
The medianRating is ****
The most frequent rating is *
```

# Using Runtime Loaded Libraries

An image that uses dynamic libraries as runtime loaded libraries is smaller and loads faster than the image using the same libraries as dependent libraries. The static linker doesn't add information about the runtime loaded libraries to the image. And the dynamic loader doesn't have to load the library's dependent libraries when the image is loaded. However, this flexibility comes at a price. Before an image can use a dynamic library that is not one of its dependent libraries, it must load the library with dlopen and get the address of each symbol it needs with dlsym. The image must also call dlclose when it's done using the library.

The StarMeals2 application provides the same functionality that StarMeals provides. But StarMeals2 uses the Ratings 1.1 dynamic library as a runtime loaded library. Listing 5 (page 71) shows the application's source code.

**Listing 5**      Using Ratings 1.1 as a runtime loaded library

```
/* File: StarMeals2.c
 * Uses functions in libRatings.A.dylib.
 **************************************/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <dlfcn.h>
#include <Ratings.h>

#define MAX_NAMES 100
#define MAX_NAME_LENGTH 30
#define MAX_RATING_LENGTH 5

static char* name_list[MAX_NAMES];
static char* rating_list[MAX_NAMES];
static int names = 0;

void addNameAndRating(char* name, char* rating) {
    name_list[names] = strdup(name);
    rating_list[names] =
        (strlen(rating) > MAX_RATING_LENGTH)?
        "*****" : strdup(rating);
```

```
        names++;
}

void test_data(void) {
    addNameAndRating("Spinach", "*");
    addNameAndRating("Cake", "****");
    addNameAndRating("Steak", "****");
    addNameAndRating("Caviar", "*");
    addNameAndRating("Broccoli", "****");
    addNameAndRating("Gagh", "*****");
    addNameAndRating("Chicken", "*****");
}

int main(int argc, char* argv[]) {
    int test_mode = 0;
    if (argc == 2) {
        if (strcmp(argv[1], "test") == 0) {
            test_mode = 1;
            printf("[start_test]\n");
            test_data();
        }
    }
    else {
        printf("Enter restaurant names and ratings in the form <name> <rating>.\n");
        printf("No spaces are allowed in the name and the rating.\n");
        printf("The rating can be * through *****.\n");
        printf("To finish, enter \"end\" for a restaurant name.\n");
        while (names < MAX_NAMES) {
            char name[MAX_NAME_LENGTH];
            char rating[MAX_RATING_LENGTH + 1];
            printf("\nName and rating: ");
            scanf("%s", &name);
            if (strcmp(name, "end") == 0) {
                break;
            }
            scanf("%s", rating);
            addNameAndRating(name, rating);
        }
        printf("\n");
    }

    if (names) {
        // Open Ratings library.
        void* lib_handle = dlopen("libRatings.A.dylib", RTLD_LOCAL|RTLD_LAZY);
        if (!lib_handle) {
            printf("[%s] Unable to load library: %s\n", __FILE__, dlerror());
            exit(EXIT_FAILURE);
        }

        // Print data entered and call libRatings.A:addRating().
        void (*addRating)(char*) = dlsym(lib_handle, "addRating");
        if (!addRating) {        // addRating is guaranteed to exist in libRatings.A.dylib
            printf("[%s] Unable to get symbol: %s\n", __FILE__, dlerror());
            exit(EXIT_FAILURE);
        }
        printf("This is the data you entered:\n");
        for (int i = 0; i < names; i++) {
            printf("%s (%s)\n", name_list[i], rating_list[i]);
```

```
        addRating(rating_list[i]);
    }

    // Print statistical information.
    char* (*meanRating)(void) = dlsym(lib_handle, "meanRating");
  if (!meanRating) {       // meanRating is guaranteed to exist in libRatings.A.dylib
        printf("[%s] Unable to get symbol: %s\n", __FILE__, dlerror());
        exit(EXIT_FAILURE);
    }
    printf("\nThe mean rating is %s\n", meanRating());

    char* (*medianRating)(void) = dlsym(lib_handle, "medianRating");
    if (medianRating) {     // Backwards compatibility with Ratings 1.0
        printf("The median rating is %s\n", medianRating());
    }
    char* (*frequentRating)(void) = dlsym(lib_handle, "frequentRating");
    if (frequentRating) {   // Backwards compatibility with Ratings 1.0
        printf("The most frequent rating is %s\n", frequentRating());
    }

    // Close Ratings library
    if (dlclose(lib_handle) != 0) {
        printf("[%s] Problem closing library: %s", __FILE__, dlerror());
    }
}

if (test_mode) {
    printf("[end_test]\n");
}
return 0;
}
```

Listing 6 (page 73) shows to compile the StarMeals2 application.

**Listing 6**        Compiling and linking StarMeals2

```
[StarMeals2]% make
gcc -std=gnu99 StarMeals2.c -o StarMeals2
```

The static linker doesn't complain about the unresolved external references in libRatings.A.dylib because it's not included at the link line. The dynamic linker resolves these references when StarMeals2 uses dlopen to load `libRatings.A.dylib`.

# Interposing Functions in Dependent Libraries

Sometimes you need to perform operations before or after a function is called to gather statistical data or to modify its inputs our outputs. For example, you may want to find out how many times a program calls a specific function to determine whether an algorithm should be optimized. However, you may not always have access to the function's source code to make the modifications. Interposition is a mechanism through which you can define your own version of a function that's defined in an image's dependent libraries. In your version, you may or may not call the original function.

> **Note:** In Mac OS X you can interpose only dependent libraries. Symbols in runtime loaded libraries cannot be interposed.

To call an interposed function from a custom definition, you use the `dlsym(RTLD_NEXT, "<function_name>")` call to get the address of the "real" function. For example, Listing 7 shows how you may write a custom version of a function defined in a dynamic library.

**Listing 7**      Interposing a function

```
char* name(void) {
    static int name_calls = 0;
    printf("[STATS] name() has been called %i times\n", name_calls);
    char* (*next_name)(void) = dlsym(RTLD_NEXT, "name");
    return next_name();
}
```

You may use interposition to adapt an existing dynamic library to your particular needs without changing its API. For example, this document's companion package includes the implementations of two dynamic libraries called Ratings and RatingsAsGrades. The Ratings library implements a star-based rating system (it can be used to tally restaurant and hotel ratings; for example, \*\*\* and \*\*\*\*\*). The RatingsAsGrades library implements a letter-based grading system, which can be used to tally student grades; for example A and C. Instead of writing a new algorithm to manage letter grades, the RatingsAsGrades library leverages the functionality of the Ratings library. Listing 8 (page 74) shows the interface and implementation of the RatingsAsGrades library.

**Listing 8**      RatingsAsGrades Interposing Ratings

```
/* File: RatingsInterposed.h
 * Interface to the RatingsAsGrages dynamic library.
 ************************************************/

/* Adds 'grade' to the set.
 *      grade: Non-NULL string. Contains zero or
 *             one of these characters:
 *             A, B, C, D, F.
 *             Examples: "", "A", "B".
 */
void addRating(char* grade);

/* Returns the median grade.
 */
char* medianRating(void);

/* Returns the most frequent grade.
 */
char* frequentRating(void);

/* File: RatingsAsGrades.c
 * Compile with -fvisibility=hidden.
 **********************************/

#include "RatingsAsGrades.h"
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
```

```
#include <dlfcn.h>

#define EXPORT __attribute__((visibility("default")))
#define MAX_STAR_RATING_LEN 6

static char* _ratingAsGrade(char* rating) {
    int rating_length = strlen(rating);
    if (rating_length > MAX_STAR_RATING_LEN - 1) {
        rating_length = MAX_STAR_RATING_LEN - 1;
    }
    char grade;
    switch (rating_length) {
        case 5:
            grade = 'A';
            break;
        case 4:
            grade = 'B';
            break;
        case 3:
            grade = 'C';
            break;
        case 2:
            grade = 'D';
            break;
        case 1:
            grade = 'F';
            break;
        default:
            grade = '\0';
    }
    char char_grade[2] = { grade, '\0' };
    return strdup(char_grade);
}

// Interpose libRatings.B.dylib:addRating.
EXPORT
void addRating(char* grade) {
    char rating[MAX_STAR_RATING_LEN] = { '\0' };
    switch (*grade) {
        case 'A':
            strcat(rating, "*");
        case 'B':
            strcat(rating, "*");
        case 'C':
            strcat(rating, "*");
        case 'D':
            strcat(rating, "*");
        case 'F':
            strcat(rating, "*");
        default:
            ;
    }
    void (*next_addRating)(char*) =
dlsym(RTLD_NEXT, "addRating");
    if (next_addRating) {
        next_addRating(rating);
    }
    else {
```

```
        printf("[%s] Fatal problem: %s", dlerror());
    }
}

// Interpose libRatings.B.dylib:medianRating.
EXPORT
char* medianRating(void) {
    char medianGrade[2] = { '\0' };
    char* (*next_medianRating)(void) =
dlsym(RTLD_NEXT, "medianRating");
    if (next_medianRating) {
        strcpy(medianGrade,
_ratingAsGrade(next_medianRating()));
    }
    else {
        printf("[%s] Fatal problem: %s", dlerror());
        exit(EXIT_FAILURE);
    }
    return strdup(medianGrade);
}

// Interpose libRatings.B.dylib:frequentRating.
EXPORT
char* frequentRating(void) {
    char frequentGrade[2] = { '\0' };
    char* (*next_frequentRating)(void) =
        dlsym(RTLD_NEXT, "frequentRating");
    if (next_frequentRating) {
        strcpy(frequentGrade,
            _ratingAsGrade(next_frequentRating()));
    }
    else {
        printf("[%s] Fatal problem: %s", dlerror());
        exit(EXIT_FAILURE);
    }
    return strdup(frequentGrade);

}
```

Notice how the `addRating`, `medianRating`, and `frequentRating` functions, modify the input and output of the definitions they shadow.

The companion-file package includes the source code of the Grades application. This application uses the RatingsAsGrades library to tally the grades of students.

Follow these instructions to build and run the Grades application:

1. Open this document's companion-file package.

2. In Terminal, perform the following commands:

(page 76) shows the output of the Grades application when ran in test mode.

**Listing 9**      Test output of the Grades application

```
[Grades]% ./Grades test
[start_test]
```

```
This is the data you entered:
Eloise  (F)
Karla   (B)
Iva     (B)
Hilaire (F)
Jewel   (B)
Simone  (A)
Yvette  (A)
Renee   (A)
Mimi    (A)

The median grade is B
The most frequent grade is A
[end_test]
```

# Logging Dynamic Loader Events

As you develop and use dynamic libraries, you may want to know when certain events occur. For example, you want to know when the dynamic loader binds a particular undefined external symbol or how long it took for an application to launch.

This article identifies the environment variables you can set and the type of dynamic loader logging they activate.

Table 1 (page 79) lists the environment variables that turn on logging by the dynamic loader.

**Table 1**    Environment variables that effect dynamic loader logging

| Environment variable | Description |
| --- | --- |
| DYLD_PRINT_LIBRARIES | Logs when images are loaded. |
| DYLD_PRINT_LIBRARIES_POST_-LAUNCH | Logs when images are loaded as a result of a dlopen call. Includes a dynamic libraries' dependent libraries. |
| DYLD_PRINT_APIS | Logs the invocation that causes the dynamic loader to return the address of a symbol. |
| DYLD_PRINT_STATISTICS | Logs statistical information on an application's launch process, such as how many images were loaded, when the application finishes launching. |
| DYLD_PRINT_INITIALIZERS | Logs when the dynamic loader calls initializer and finalizer functions. |
| DYLD_PRINT_SEGMENTS | Logs when the dynamic loader maps a segment of a dynamic library to the current process's address space. |
| DYLD_PRINT_BINDINGS | Logs when the dynamic loader binds an undefined external symbol with its definition. |

# Document Revision History

This table describes the changes to *Dynamic Library Programming Topics*.

| Date | Notes |
|------|-------|
| 2006-11-07 | Added information on using libtool to create libraries and on locating external resources using file-path macros. |
| | Added details about `@executable_path` and `@loader_path` in "Locating External Resources" (page 24). |
| | Added tip on using libtool to build dynamic libraries in "Setting the Library's Version Information" (page 53). |
| 2006-09-05 | Added information about the creation and usage of private embedded frameworks. |
| | Added information to "Using Dynamic Libraries" (page 67) > "Installing Dependent Libraries" (page 67). |
| 2006-02-07 | Made minor correctness changes. |
| | Removed extraneous code lines from Listing 3 (page 41). |
| 2005-08-11 | Corrected errors and typos. |
| | Changed `LS_LIBRARY_PATH` to `LD_LIBRARY_PATH` in "Using Dynamic Libraries" (page 67). |
| | Changed `DYLD_PRIINT_STATISTICS` to `DYLD_PRINT_INITIALIZERS` and corrected description for `DYLD_PRINT_SEGMENTS` in "Logging Dynamic Loader Events" (page 79). |
| 2005-06-04 | New document that shows how to correctly design, implement, and use dynamic libraries. |

82