

# **High Performance Programming for OS X**

**Timothy Ritchey**

**High Performance Programming for OS X**  
by Timothy Richey

# Table of Contents

<b>1. Under the Hood with Objective-C</b>	<b>1</b>
1.1. Objective-C's Family Tree	2
1.2. Why Objective-C?	3
1.3. Objective-C Programming in OS X	4
1.3.1. Project Builder	4
1.3.2. The Compiler	4
1.3.3. Objective-C API's	5
1.3.3.1. Foundation Kit	5
1.3.3.2. AppKit	5
1.3.3.3. CoreFoundation	5
1.3.3.4. Cross-platform Objective-C APIs: GNUStep	6
1.4. The Objective-C Runtime	7
1.4.1. Creating Objects	7
1.4.1.1. What isa Class	9
1.4.1.2. Class Information	12
1.4.1.3. Instance Variables	14
1.4.1.4. <b>class-dump</b>	16
1.4.1.5. <code>alloc</code>	17
1.4.2. Talking to Objects	18
1.4.2.1. Key-Value Coding	24
1.4.2.2. Optimizing with IMPs	25
1.4.3. Destroying Objects	28
1.4.3.1. Objective-C Reference Counting	28
1.4.3.2. Autorelease Pools	29
1.5. Back Together Again: Objective-C++	29
1.5.1. Using C++ Classes from Objective-C	32
1.5.2. Using Objective-C Classes from C++	38
1.5.3. Things to Watch Out For	44
1.6. Resources	45

# List of Tables

1-1. <code>ivar_type</code> String Values for Objective-C Types.....	14
--	----

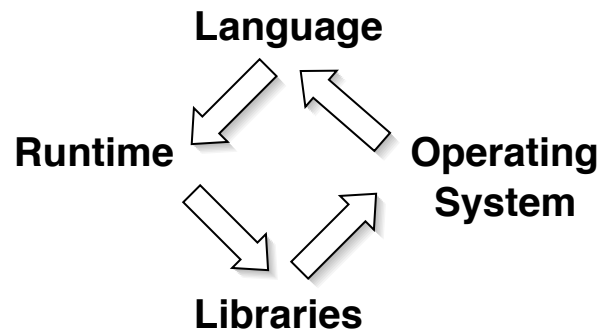
# List of Figures

1-1. Interacting With OS X.....	1
1-2. Carbon, Cocoa and CoreFoundation .....	6
1-3. <code>foo_bar</code> Versus <code>FooBar</code> Memory Layout.....	8
1-4. The <code>objc_class</code> Structure.....	10
1-5. Classes and Objects .....	12
1-6. Lignarius Cut List Tab.....	17
1-7. Sending a Message With <code>objc_msgSend( )</code> .....	23
1-8. ProjectBuilder File Type for Objective-C++ .....	29
1-9. Calling C++ from Objective-C.....	33
1-10. A Simple Table View.....	34
1-11. Create an Instance of our Objective-C++ Object <code>DataSource</code> .....	35
1-12. Create an Instance of the <code>WoodCountController</code> Class.....	37
1-13. Our New Mixed Objective-C and C++ Application Running .....	38
1-14. Integrating Objective-C into a C++ Program .....	39

# Chapter 1. Under the Hood with Objective-C

Just like knowing where the gas and brake pedal are located doesn't automatically make you a formula-one driver, knowing how to program in any particular language doesn't necessarily give you the wherewithal to create high performance programs. Sometimes you have to get under the hood to get a good idea of what it takes to go fast.

Superficially, Objective-C is a very easy language to learn and use. Of course, nothing comes for free. The simplicity of programming for OS X using Objective-C masks the various elements that must come together to make it all work.

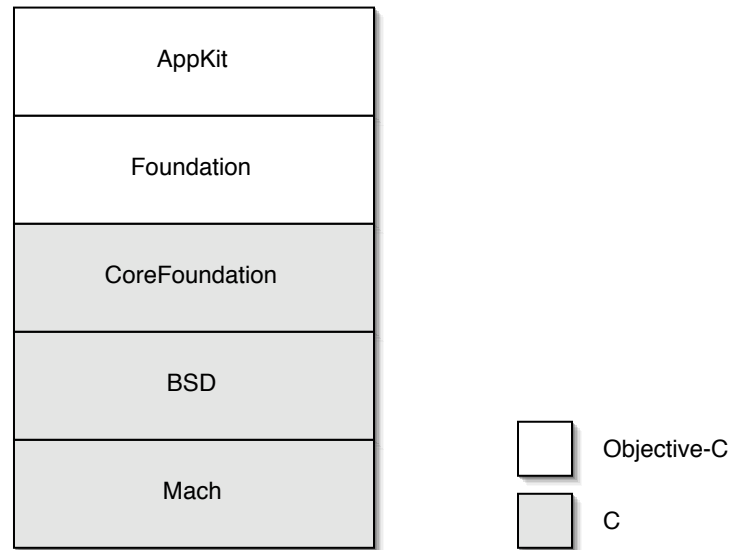


Of course, the component we are most cognizant of is the language itself—it is what confronts us continuously in our editor windows. Hopefully you are already familiar with the Objective-C language. If you have come to this book from a non-Objective-C background, Section 1.6, at the end of the chapter provides several good sources for learning Objective-C.

Languages, no matter how good, do not stand on their own. To develop meaningful applications, languages must be able to interact with system resources. Additionally, depending upon its complexity, certain features of a language may require a runtime system, as is the case with Objective-C. Many of Objective-C's features that make it stand out from other languages are made possible by its runtime.

Just as critical as a language's syntax and semantics are the available libraries. Ultimately, libraries are how real work gets done in any language. Otherwise, programmers would have to create their own code for interacting with system resources such as networks, storage, input devices and displays. With Objective-C and OS X we have a plethora of choices when it comes to using libraries. This variety comes from Objective-C's ability to seamlessly use C, Objective-C, and even C++ (See Section 1.5). This variety of language options means that Objective-C programmers can interact with OS X at many different levels. Figure 1-1 shows some of the many faces of OS X available to Objective-C.

At the bottom (or the top, depending on how you look at it) of the stack is the *Operating System*. The OS is the environment within which all of this is possible. You have undoubtedly been forced to use, at one time or another, operating systems that were less than robust<sup>1</sup>—that perhaps felt more like an unsteady deck of cards, than a fine tuned piece of software. Thankfully, most operating systems these days do not fall into this category. Understanding how these modern operating systems work is important to understanding how your applications will perform when running on top of them.

**Figure 1-1. Interacting With OS X**

In the next few chapters, we will descend through these different layers of programming in Objective-C on OS X, focusing on the implementation details that will give you an intimate understanding of what is going on under the hood. This chapter focuses on the underlying mechanisms of the Objective-C language and runtime. covers the use of system and third-party libraries, as well as issues involved in creating your own. will take a more detailed look at the OS X operating system, including both its benefits and quirks. The last part of this book will apply all of this information to specific techniques you can use to create high performance applications.

## 1.1. Objective-C's Family Tree

Every Thanksgiving my family would take a 500 mile trip south to my grandparents in Tennessee. When we finally stumbled out of the car, dazed and sickened from exhaust fumes, roller-coaster hills and eight hours of eight-track music, everything always seemed just a little weird. Of course, being from Illinois, we were considered the "Yanks." Everyone spoke with an unintelligible accent, and there were always a few strange, toothless old men sitting around on the porch, drinking iced tea from mason jars, spitting tobacco, and otherwise having a high old time.

Coming from C++, this about describes the feeling I had upon looking at Objective-C code for the first time. It was disconcerting at first, but very quickly you come to realize you are among family. Of particular consternation was the use of brackets to perform, what I thought at the time were, C++'s version of object method calls:

```
NSMutableArray *cutList = [[NSMutableArray alloc] init];
```

*Why go to all the trouble?* I thought to myself. It wasn't until I came to understand the difference in the object systems for C++ and Objective-C that the syntax began to make sense.

Both Objective-C and C++ were originally conceived as object oriented extensions to C. C++ was written by Bjarne Stroustrup and comes from the Simula 67 school, which places more import on compile-time type safety over run-time dynamism. The syntax of C++ sticks closer to that of C proper, and is built to make user-defined classes act as much as possible like the built-in types of C. Features like operator overloading allow programmers to write

classes that intuitively feel like their keyword counterparts. For numerical programming, this allows for great elegance. One need only look at the plethora of matrix classes that allow operations such as:

```
A += B * C;
```

to see why C++ has so many proponents. In addition, Generic Programming and the Standard Template Library (STL) in particular, have pushed forward the idea of what can be encapsulated in code. Libraries such as Andrei Alexandrescu's pattern templates, which provide the ability to concretely specify and implement design patterns, and the Graph Boost Library, which provides generic graph storage and algorithm templates, are excellent examples of the power of generic programming. Without a doubt, for hard core number crunchers, C++ has some great advantages.

Objective-C was written by Brad Cox in order to add smalltalk-80 extensions to C, and had not experienced widespread acceptance in either the operating system or application developer community. The one place Objective-C *was* picked up happened to be Steve Jobs post-Apple computer company NeXT. When Apple purchased NeXT in order to acquire what would later become OS X, it also inherited Objective-C.

Objective-C is fundamentally a dynamic language. That is, almost all decisions regarding object type and method calls are made at run-time, as opposed to compile-time. Many aspects of Objective-C and Cocoa that make OS X such a pleasant platform to program are derived from this dynamism. Tools such as Interface Builder and technologies like Distributed Objects get much of their power from the ability to handle completely new objects without undue overhead. While techniques exist in many forms on different platforms (such as .NET or CORBA), Objective-C has a twofold advantage: First, Objective-C is still C, retaining the lower-level speed and efficiency traditionally associated with the language. Second, Objective-C's object-oriented extensions are simple--the set of syntactic changes are small compared to the power they offer a programmer.

Unlike C++, which seems like it is trying to hide its object-oriented nature behind traditional C syntax, Objective-C's terminology and syntax are meant to emphasize the difference between objects and C's traditional procedural paradigm. Seeing

```
[cutList addObject:newPiece]
```

immediately shouts to us that `cutList` is an object<sup>2</sup>. When you see the brackets, you know you are doing something radically different from typical C code. In this case, you are interacting with a dynamic run-time system to pass messages between objects.

## 1.2. Why Objective-C?

If C++ has so many advantages, why would we choose to ever use Objective-C? To start, for an object oriented language, Objective-C is small. Apart from getting used to Objective-C's unusual bracket syntax, most programmers familiar with C and another object-oriented language can pick up Objective-C very quickly. For those programmers new to object-oriented programming altogether, which is becoming an ever smaller group, Objective-C is a very good language with which to learn OOP principles.

At the same time that Objective-C is quick to learn, combined with Cocoa, it allows developers to create complex graphical applications more quickly. Personally, I find myself completing applications or tools that I would have *never* considered tackling by myself on Windows or Linux. By using a simpler language, you will find yourself spending less time fixing typos and double checking syntax peculiarities, and more time getting your projects done.

In fact, this should be the first rule of high performance programming:

The program that is written, *no matter how slow*, will always finish faster than the program that is never completed.

Many hours of programmer productivity have been lost to premature optimization—which you will hear about time and time again in any discussion about high performance programming. Unfortunately, it is every programmer's right to learn this lesson the hard way. One of the first optimization decisions we make is choosing the language in which to write our application's code. Choosing a language presents us with the classic trade-off—do we choose a language that makes the program easier to write, or do we choose the language that will result in the fastest run-time? Objective-C is able to provide a RAD-like development environment, while retaining its low-level C functionality. This means you can first concentrate on getting your application built, and then worry about making it fast. In many ways, this follows the development of OS X itself.

As a final point, if you are still struggling with the choice between the ease of Objective-C and the power of C++, worry not. Apple has provided a version of Objective-C that can interact easily with both called Objective-C++. This language version will be presented in more detail in .

## 1.3. Objective-C Programming in OS X

Apple's primary development environment for Objective-C is Project Builder. This integrated development environment provides all of the traditional graphical tools programmers have come to expect, including editing, class browsing, compiling and debugging. Being Unix, you can imagine that command line tools are not far away either. In fact, the underlying compiler and debugger will, more than likely, be very familiar to you. Finally, the development tools available from Apple include a number of ancillary utilities for handling resources, packaging applications, and profiling applications.

### 1.3.1. Project Builder

I would expect most readers to be intimately familiar with Project Builder, and use it in their day-to-day OS X application development. There are other options, of course, including Metrowerk's Code Warrior, or for the real hackers among us: **emacs** or **vi**, with **gcc** and **gdb** from the command line.

For the rest of us mere mortal programmers, IDEs like Project Builder bring all of the most useful tools together in one place, and provide pre-packaged project templates that isolate us from complex build scripts. In most cases, this is a good thing. The less time you spend debugging **make** files, the more time you can spend debugging your own code. At the same time, the number of questions regarding missing libraries on the Cocoa mailing lists indicates that many new programmers are so used to IDEs doing all the work, that they may not fully understand what is going on underneath. The next several chapters should hopefully bring you up to speed.

### 1.3.2. The Compiler

Apple provides an excellent development environment for OS X in Project Builder, especially considering that it is available for free. Under the hood though, everything is running on open source products. This includes the **gcc** compiler, a debugger **gdb**, and other miscellaneous tools such as **make**. As of writing, the latest tool chain delivered with OS X 10.2 is based on **gcc** 3.1. ??? get latest before publish ??? Prior to this release, Project Builder used **gcc** 2.95.2.

If you are moving from OS X 10.1 to 10.2, with an attendant change in **gcc** from 2.95.2 to 3.1, there are fundamental changes in certain areas that can make libraries incompatible. In particular, the C++ ABI was changed for **gcc** 3.0+, which makes it incompatible with earlier versions. If you have recently upgraded, and are having problems with a project linking,



make sure you clean your project by selecting: **Build**→**Clean** from Project Builder's menu. It is a good idea any time you upgrade your tools to clean and rebuild your projects.

Apple's use of the FSF tools is not a one-way street. Apple has its own developers working on patches and updates to **gcc**, mostly involving changes to better support OS X and the PowerPC processor. These updates are fed back into the **gcc** project. Not all of this code is incorporated into the "official" **gcc** releases right away, so be sure you check to make sure all the features you are using are available across the platforms in which you might be interested. One example of this lag is in support for Objective-C++. Apple has submitted it for inclusion, but as of version 3.1, the official **gcc** release does not include Objective-C++ support. You will only be able to find Objective-C++ in Apple's distribution of the tools for the time being.

We will leave **gcc** for the time being; however, we will return to it in where we will take a closer look at how to use its optimization features.

### 1.3.3. Objective-C API's

OS X's Objective-C API is called `Cocoa`. `Cocoa` is basically the OS X port of the original NeXTStep API. This is apparent when you see the names assigned to all the classes, such as `NSString`, `NSWindow` and `NSView`. The `NS` stands for NeXTStep. NeXT later called this API `OpenStep`, when it was ported to additional platforms, such as Solaris and Windows NT, but kept the `NS` designator. OS X continues this tradition.

`Cocoa` supports both Objective-C and Java as languages, although it is more traditionally considered an Objective-C based API. The API itself is separated into two frameworks: `Foundation` and `AppKit`. If you

```
#import <Cocoa/Cocoa.h>
```

, you are in fact including both frameworks. This is what is known as an *Umbrella Framework*.

#### 1.3.3.1. Foundation Kit

`Foundation Kit` is the framework that provides all of the core, non-graphical classes for `Cocoa`, as well as many structures and stand-alone functions. Examples of typical `Foundation` classes are `NSSet`, `NSArray` all the way up the hierarchy to `NSObject`. If you want to program at any level, graphical or not, in Objective-C, you must include the `Foundation` framework.

#### 1.3.3.2. AppKit

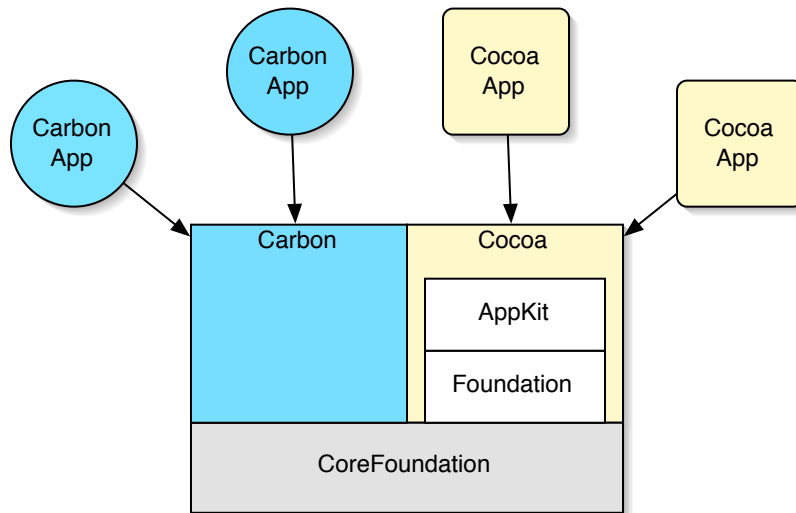
`AppKit` is the framework that provides all of the elements necessary for creating graphical applications with `Cocoa`. Primarily, this means all of the GUI elements, but also includes classes for supporting different kinds of applications, such as single window or document-based.

If you know you will not be referring to `AppKit` classes in a source file, you can save some compile time by only importing the `Foundation` framework. Project Builder does this by default when you select **File**→**New File...**, and choose the **Objective-C Class** option. These kinds of optimizations are covered in more detail in

#### 1.3.3.3. CoreFoundation

Apple is in the unfortunate position of needing to support two developer bases: old Macintosh programmers, and old NeXT programmers. Dealing with this split personality has consumed an amazing amount of programming effort on Apple's part, and generated very heated debates among Macintosh and NeXT devotees. The end result of all this

Figure 1-2. Carbon, Cocoa and CoreFoundation



activity is a three pronged approach in OS X. For existing MacOS 9 and earlier applications, OS X provides a “Classic” environment in which they can run. If you want your application to run natively, you can either use the more Mac-like Carbon C API, or the NeXT-like Cocoa Objective-C API. Before the client version of OS X was released, Apple made a fundamental change to both the Carbon and Cocoa implementations. Instead of two entirely separate APIs and attendant libraries, Apple developed a CoreFoundation framework that implements many of the common features of both systems. Carbon and Cocoa can now both call into this common library, allowing Apple to more easily maintain and reuse code between the two APIs (See Figure 1-2).

You can recognize CoreFoundation methods and structures by their `CF*` designation. While CoreFoundation is entirely C-based, its implementation and structures are designed in a very object-oriented-like manner. Many Foundation objects can be directly mapped to their implementations in CoreFoundation. `CFDictionary` and `NSDictionary` are a good example of this pairing.

While the source code for Carbon and Cocoa themselves are not available, Apple has decided to redistribute CoreFoundation a part of the underlying Darwin Open Source project. This means all of the source code for CoreFoundation is available for us to study. Because so many of the fundamental data structures we use every day in our applications are implemented here, understanding how they work is important when it comes time to figuring out where performance improvements can be made in our code.

#### 1.3.3.4. Cross-platform Objective-C APIs: GNUStep

When talking about cross-platform support for Objective-C we are really talking about the NextStep API. Since `gcc` is available on essentially every platform, the Objective-C language and runtime itself is also available. Of course, this cross-platform availability does us no good if the libraries we use to create our applications are not available as well. GNUStep is a project to deliver an open source implementation of the OpenStep API that can be run on all the major platforms for which `gcc` is available.

GNUStep is a work in progress. Foundation and Distributed Object support exists, and is in very good shape. AppKit support is a little spottier. While it is nearly feature complete on UNIX derivatives—including OS X itself, its Win32

port is somewhat behind the curve. This means, for the time being, cross-platform Objective-C programming does not fully address Win32, to which unfortunately, is what platform most developers are hoping to port. Hopefully, by the time this reaches bookshelves, the situation will be somewhat improved. If you are targeting OS X and Unix platforms only, then GNUStep is able to provide a great cross-platform solution. We will return to GNUStep when we develop a cross platform distributed application in

## 1.4. The Objective-C Runtime

Objective-C's ease-of-use, flexibility and power come at a price. As you can guess, that price is almost always in performance. Instead of calculating class types and function addresses at compile-time, Objective-C waits to perform these lookups when they are actually called on at run-time. This is unfortunately the same precious time we are trying to conserve in our programs. Object-oriented programming is foremost a tool to speed the *process* of programming, not necessarily a programs processing speed.

There is room for optimism, however. As we have mentioned before, Objective-C is still C. When you need to put the petal to the metal, you can always fall back on one of the most efficient languages around. In addition, as long as you know what the Objective-C runtime is doing, you can learn how to use it to best effect.

If you stick with the C in Objective-C, many of the traditional performance techniques apply. As soon as you put brackets around something in Objective-C, you intuitively know a whole host of new object-oriented issues must be addressed. These issues can be generally broken down into three broad themes: Object Creation, Message Passing and Object Destruction. This section presents these themes, and shows how consideration of object lifecycle is of paramount importance to high performance object oriented programming.

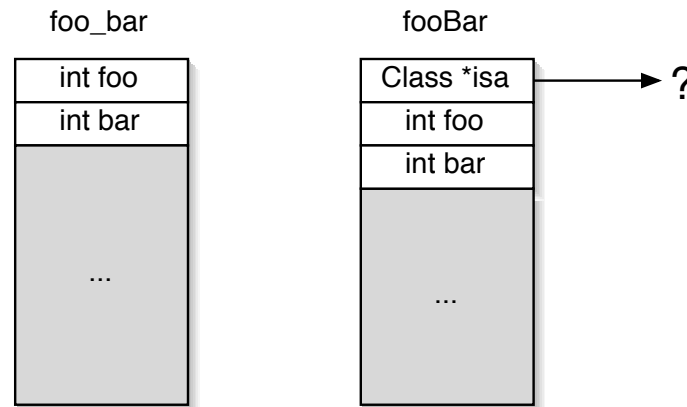
When talking about an object's lifecycle, we are referring to everything that happens from the time it is created to the time it is destroyed. Just like real life objects, every object in your program consumes environmental resources while being made, during its lifetime, and finally being destroyed. These resources may be processor clock cycles, memory usage, or I/O with the network or a disk. No matter what the case, ultimately it is a consumption of the most scarce resource of all: time.

### 1.4.1. Creating Objects

Given the following trivial class interface:

```
@interface FooBar : NSObject {
    int foo;
    int bar;
}
- (id)initWithFoo:(int)f bar:(int)b;
- (int)foo;
- (void)setFoo:(int)f;
- (int)bar;
- (void)setBar:(int)b;
@end
```

Even though you might not know what any particular class or method name of `FooBar` means, you probably won't have any problem reading the following code:

Figure 1-3. `foo_bar` Versus `FooBar` Memory Layout

```

FooBar* fooBar = [[FooBar alloc] initWithFoo:21 Bar:12];

NSLog(@"fooBar's Foos:%d Bars:%d", [fooBar foo], [fooBar bar]);

[fooBar setBar:3];

NSLog(@"fooBar's Foos:%d Bars:%d", [fooBar foo], [fooBar bar]);

[fooBar release];

```

As Objective-C programmers, we have a good feeling for the syntax of the language. What most Objective-C programmers don't know is what exactly the objects they are throwing around in code actually *are*. If I present you with the following structure:

```

struct _foo_bar {
    int foo;
    int bar;
} foo_bar;

```

you probably have a good feeling for what `foo_bar` looks like in memory. When we create the Objective-C object `fooBar` above, we know we receive a pointer to a memory location, but what does that memory location look like? It turns out that Objective-C objects aren't much different than C's structures. Figure 1-3 shows us how both the C structure and Objective-C object are laid out in memory.

The only difference between the structure and object in memory is the prefix of the `isa` pointer at the beginning of the memory location. It is this pointer that provides the Objective-C runtime all the information it needs to handle the object. The following code explicitly shows how we can create our own Objective-C object by hand:

```

typedef struct _foo_bar {

```

```

    Class isa;
    int foo;
    int bar;
} foo_bar;

int main (int argc, const char * argv[]) {
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

    // traditionally created object
    FooBar *fooBar = [[FooBar alloc] initWithFoo:12 bar:21];

    // creating our own
    foo_bar* fb = malloc(sizeof(foo_bar));
    fb->isa = (Class*)[FooBar class];
    fb->foo = 6;
    fb->bar = 2;

    // use the alloc'ed FooBar object fooBar
    NSLog(@"fooBar's Foos:%d Bars:%d", [fooBar foo], [fooBar bar]);
    [fooBar setBar:3];
    NSLog(@"fooBar's Foos:%d Bars:%d", [fooBar foo], [fooBar bar]);
    [fooBar release];

    // use our handcrafted foo_bar object fb
    NSLog(@"fooBar's Foos:%d Bars:%d", [fb foo], [fb bar]);
    [fb setFoo:62];
    [fb setBar:26];
    NSLog(@"fooBar's Foos:%d Bars:%d", [fb foo], [fb bar]);

    [pool release];
    return 0;
}

```

In this case, we create a structure `foo_bar` that explicitly holds our `Class` pointer `isa`, as well as our two `ints` `foo` and `bar`. In order to use our structure, we need to allocate memory for it, which we do using the traditional `malloc()` system call. We then fill in the structure with a pointer to `FooBar`'s class, as well as initialize the two integer variables. Once we go to the trouble, we can use our made-from-scratch `foo_bar` just like it was a proper Objective-C object<sup>3</sup>.

You might have run across the term “Toll-free Bridging” in reference to structures in the CoreFoundation library that can be freely interchanged with objects in Objective-C code. One example of this kind of object is `CFDictionary` and `NSDictionary`. If a CoreFoundation function call requires an object of type `CFDictionary`, you can pass in an `NSDictionary` unaltered. You can see why this is easy for the Apple engineers to support—the C structure is layed out identical to the Objective-C object. The C code merely ignores the `isa` pointer at the beginning.

### 1.4.1.1. What isa Class

Up to this point, we have referred to the `FooBar` class somewhat cavalierly, even calling `[FooBar class]` without explanation. What *is* `isa` pointing to? If you look in `/usr/include/objc/objc.h` you will find that `Class` is defined as: `typedef struct objc_class *Class;`. So, `Class` is a pointer to an `objc_class` structure, which is defined in `/usr/include/objc/objc-class.h` as:

```
struct objc_class {
    struct objc_class *isa;
    struct objc_class *super_class;
    const char *name;
    long version;

    long info;
    long instance_size;
    struct objc_ivar_list *ivars;
    struct objc_method_list **methodLists;
    struct objc_cache *cache;
    struct objc_protocol_list *protocols;
};
```

So, for every class, the Objective-C runtime system creates a `Class` object based on the `objc_class` structure. This structure provides all of the necessary bookkeeping for our instance objects. This bookkeeping includes the class name, a pointer to its superclass, versioning information, as well as pointers to structures holding method and protocol information. Figure 1-4 shows the layout of the `objc_class` structure.

Note that even the `Class` structure is a proper Objective-C object, holding its own `isa` pointer. For every class, such as `FooBar`, the Objective-C runtime creates a `Class` object. It is this object which responds to the class (as opposed to instance) methods we define.

When you create a class such as:

```
@interface FooBar : NSObject {
}
+ fooBarWithFoo:(int)f bar:(int)b;
@end
```

it is the `FooBar Class` object that receives the message when you call:

```
FooBar *fb = [[FooBar fooBarWithFoo:12 bar:34] retain];
```

Objective-C keeps track of this object just like every other object in the system, with its own `objc_class` structure pointed to by `isa`. This `Class` object's `Class` is considered the *metaclass* for `FooBar`. Just as our `FooBar Class`

Figure 1-4. The objc\_class Structure

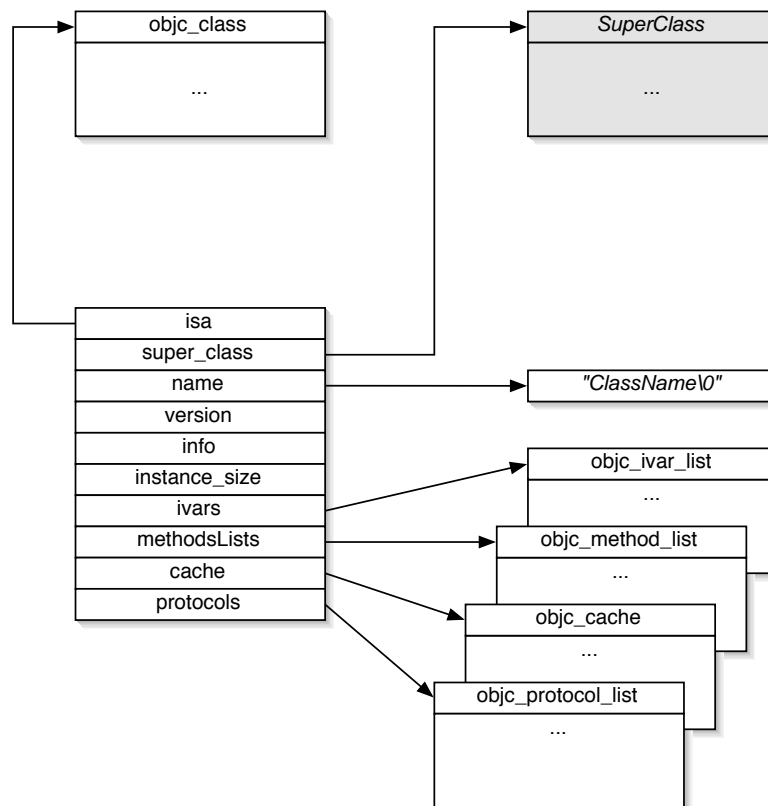
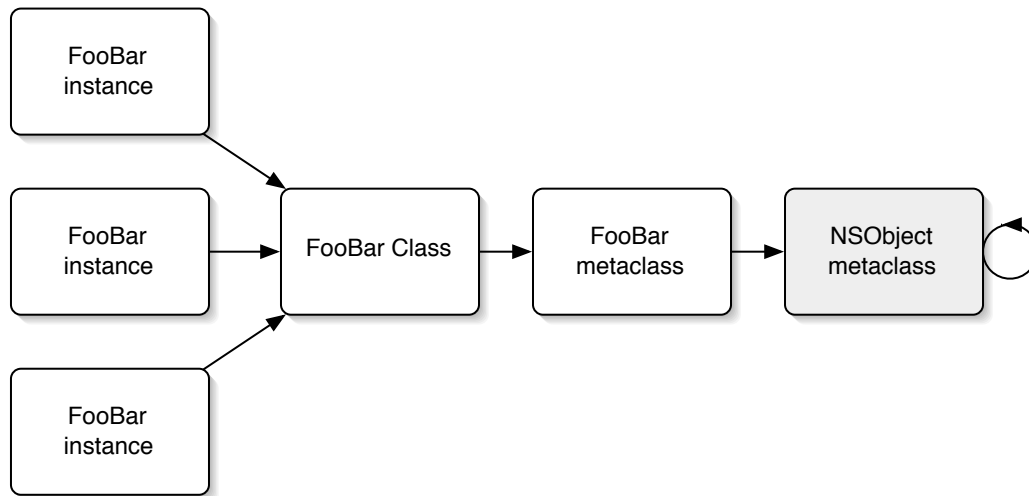


Figure 1-5. Classes and Objects



keeps track of *instance* methods, the `FooBar` metaclass keeps track of *class* methods. You can see this relationship in Figure 1-5.

It is important to press home an often misunderstood detail at this point. For each class, there is only *one* `Class` object. This single `Class` object is often referred to as a *factory*. That is, it is a factory that knows how to create objects of a particular type. This notation even turns up in `gcc` error messages like:

```
cannot find class (factory) method
```

All objects of a specific class type point to the same `Class` (or factory) object through their `isa` member. This emphasizes the difference we mean when we refer to a *class* versus an *object*. There is only one `FooBar Class`, while there can be any number of `FooBar` objects (See again Figure 1-5).

All of this class-level information is contained behind the single `isa` pointer that sits at the beginning of each class, and is maintained by the Objective-C runtime. For creating objects, it is left to `alloc` and `init` to prepare the instance variables that make each object unique.

#### 1.4.1.2. Class Information

The Objective-C runtime maintains a wealth of information about a class in the `objc_class` structure. At the most basic level is data regarding the class name, version, etc.. The following are the declarations for these `objc_class` structure elements:

```
const char *name;
long version;

long info;
long instance_size;
```



The name member is what you would expect—a c-string representation of the class name as read by the compiler in the original class @interface declaration.

Objective-C supports the concept of versioning at the class-level, which is expressed by the existence of a version member of the objc\_class structure. NSObject provides a couple class methods for setting the version number of a class called: +setVersion: and +version. +setVersion: is called in the class' +initialize class method:

```
+ (void)initialize
{
    [Foo setVersion:2];
}
```

Class versioning has typically been used to provide backward compatibility for archiving objects. By using the +version method, un-archiving code can detect the original version of the serialized class, and restore it appropriately.

The objc\_class info variable is used to flag certain class conditions. For the curious, the Flags are defined in /usr/include/objc/objc-class.h as:

```
#define CLS_CLASS    0x1L
#define CLS_META    0x2L
#define CLS_INITIALIZED  0x4L
#define CLS_POSING    0x8L
#define CLS_MAPPED    0x10L
#define CLS_FLUSH_CACHE  0x20L
#define CLS_GROW_CACHE  0x40L
#define CLS_NEED_BIND  0x80L
#define CLS_METHOD_ARRAY    0x100L
// the JavaBridge constructs classes with these markers
#define CLS_JAVA_HYBRID  0x200L
#define CLS_JAVA_CLASS  0x400L
// thread-safe +initialize
#define CLS_INITIALIZING 0x800
```

Unless you are hand coding custom classes, or working on the Objective-C runtime, you will probably not have any need for the info variable.

The instance\_size member of the objc\_class structure holds the number of bytes an instance of the class requires in memory. This size includes the 4 bytes required for the isa pointer at the beginning. Given the following classes:

```
@interface Foo : NSObject {
    int a;
    double b;
}
@end

@interface Bar : Foo {
    NSString *c;
}
```

```
@end
```

the `instance_size` for `Foo` will hold the value 16—four bytes for the `Class` pointer `isa`, which is inherited from `NSObject`, four bytes for the `int a` and eight bytes for the `double b`. The `instance_size` for `Bar` is 20. The size of `Bar` includes the 16 bytes required for `Foo`, plus the additional four bytes for the `NSString` pointer. The `instance_size` variable is inclusive of all superclass requirements.

### 1.4.1.3. Instance Variables

The `objc_class` structure also keeps track of your class' instance variables in the `objc_ivar_list` structure `ivars`. This structure is declared in `/usr/include/objc/objc-class.h`.

```
struct objc_ivar_list {
    int ivar_count;
#ifdef __alpha__
    int space;
#endif
    struct objc_ivar ivar_list[1];          /* variable length structure */
};
```

**Tip:** Note the use of the `#ifdef __alpha__` conditional construct. This makes sure the `ivar_list` array is aligned on an eight-byte boundary, or 64-bits. We will take a closer look at alignment in .

The `objc_ivar_list` structure holds a variable length array of `objc_ivar` structures:

```
typedef struct objc_ivar *Ivar;

struct objc_ivar {
    char *ivar_name;
    char *ivar_type;
    int ivar_offset;
#ifdef __alpha__
    int space;
#endif
};
```

The `objc_ivar` structure holds information on each instance variable declared in a class' `@interface`. This information includes the variable's name, its type, and its offset within an instantiated class' memory layout. The variable type is encoded as a C-string. For Objective-C types, Table 1-1 shows the relevant codes.

Pointers are prefaced with a caret (^), so that `void *` would return a type string of `^v`. Objective-C classes are encoded as their full class names. An `NSString` instance variable would return a type of `@NSString`.

When presenting type information for structures or unions, the `ivar_type` variable is a little more verbose. The following are the type output for the `NSRect` structure, and a union comprised of an `int x` and a `double y`:

**Table 1-1. `ivar_type` String Values for Objective-C Types**

Type	Label
void	v
char	c
unsigned char	c
short	s
unsigned short	S
int	i
unsigned int	I
long	l
unsigned long	L
float	f
double	d
bit field	b
pointer	^
char*	*
id	@
Class	#
SEL	:
IMP	^
BOOL	c
undefined	?

```
{_NSRect="origin" {_NSPoint="x"f"y"f} "size" {_NSSize="width"f"height"f}}
(?="x"i"y"d)
```

### Class Variables

If you are familiar with C++, you will probably have run across the use of the static keyword to define class variables.

Objective-C does not have the concept of class variables. Instead, Objective-C uses static variables at *file* scope to simulate class variables.

#### 1.4.1.4. class-dump

The dynamic nature of Objective-C leads to some interesting side-effects. Because the structures of classes, protocols and categories are included in a program's object files in human readable form, this information is available to anyone wanting to take a look. **class-dump**<sup>4</sup>, by Steve Nygard, is a useful utility for examining the Objective-C segment of Mach-O files. See the section entitled "Mach-O" in for more information on the Mach-O file format.

class-dump 2.1.5 Usage:

```
class-dump [-a] [-A] [-e] [-R] [-C regex] [-r] [-s] [-S] executable-file
```

```
-a  show instance variable offsets
-A  show implementation addresses
-e  expand structure (and union) definition whenever possible
-R  recursively expand @protocol <>
-C  only display classes matching regular expression
-r  recursively expand frameworks and fixed VM shared libraries
-s  convert STR to char *
-S  sort protocols, classes, and methods
```

here is a small sample of class-dump's output on Lignarius:

```
@interface CutListController:NSObject {
    NSTextField *countTextField;
    NSTextField *widthTextField;
    NSTextField *heightTextField;
    NSTextField *depthTextField;
    NSPopUpButton *scalePopUpButton;
    NSTextField *descriptionTextField;
    NSPopUpButton *materialPopUpButton;
    NSComboBox *speciesComboBox;
    NSPopUpButton *grainPopUpButton;
    NSButton *updateButton;
    NSButton *deleteButton;
    NSButton *addButton;
    NSTableView *cutListTableView;
```

Figure 1-6. Lignarius Cut List Tab

Count	Material	Species	W	H	D	Grain	Description
12	Solid	Cherry	24.00	2.00	0.75	Width	Face Frame Rails

```

WoodSelectionController *woodSelection;
NSMutableArray *cutList;
}
- init;
- (void)awakeFromNib;
- (void)addPiece:fp12;
- (void)woodSelectionMaterialChanged:fp12;
- (void)woodSelectionSpeciesChanged:fp12;
- (void)updatePiece:fp12;
- (void)deletePiece:fp12;
- (void)clearForm:fp12;
- (void)materialSelected:fp12;
- (int)numberOfRowsInTableView:fp12;
- tableView:fp12 objectValueForTableColumn:fp16 row:(int)fp20;
- (void)tableViewSelectionDidChange:fp12;
- (void)dealloc;

@end

```

This object is the controller for the window shown in Figure 1-6

**class-dump** is a handy tool when you want to take a peek at what objects make up an application, and how those objects operate.

#### 1.4.1.5. alloc

When we subclass NSObject, we inherit the `alloc` and `allocWithZone:` methods. These are the standard methods that allocate the memory our object needs, and set the `isa` pointer in the object's class structure. For most Cocoa programming, `alloc` and `allocWithZone` are all you will need.

While the default `alloc` methods we inherit from `NSObject` do their jobs well, they can't make the kind of optimization assumptions we can—if we have a good idea how our objects will be used. It frequently turns out that object creation (along with object destruction) is one direction we can turn in order to speed up our programs. In we will take a more in depth look at object creation and memory usage by Objective-C objects for optimization purposes.

### 1.4.2. Talking to Objects

Creating objects is not much different than what we have always done in C to manage data structures. The real power of OOP, however, comes from the encapsulation of data behind an interface of well defined methods. While the Objective-C object structure provides the necessary bookkeeping for associating objects with their methods, it is the Objective-C runtime that makes it all possible. Passing a message to an Objective-C object is much different than calling a C function.

Consider the traditional C methodology for calling a function. The following code calls a simple function named `foo()`:

```
#include "stdio.h"

int foo(int bar)
{
    return bar + 12;
}

int main() {
    int bar = foo(12);
    printf("foo:%d\n", bar);
    return 0;
}
```

We can take a look at the compiled code disassembled in **`gdb`** using the following command:

```
(gdb) disass main
```

While pouring over assembly is becoming an ever rarer occurrence in modern day computer programming, it is still useful in optimization and high performance applications. If you are not familiar with using **`gdb`** or the PowerPC assembly language, there is an introduction to **`gdb`** in [and](#) PowerPC assembly in [. Later, we will look more closely at AltiVec assembly in  \[and \\[.\\]\\(#\\)\]\(#\)](#)

The following assembly is the first portion of **`gdb`**'s dump of the `main()` routine, right up until the point it calls `foo()`.

```
0x1e74 <main>: mflr    r0
0x1e78 <main+4>: stmw    r30,-8(r1)
0x1e7c <main+8>: stw     r0,8(r1)
0x1e80 <main+12>: stwu    r1,-80(r1)
0x1e84 <main+16>: mr      r30,r1
0x1e88 <main+20>: bcl-    20,4*cr7+so,0x1e8c <main+24>
0x1e8c <main+24>: mflr    r31
0x1e90 <main+28>: li      r3,12
0x1e94 <main+32>: bl      0x1e48 <foo>
```

0x1e98 ...

We are interested in the very last line, where `foo()` is called with `bl 0x1e48`. `bl` is the PowerPC assembly mnemonic for *Branch and Link*. This command tells the processor to jump to location `0x1e48` in the program, after saving the next address after the branch code in the link register—in this case `0x1e98`. After the return address is saved, the processor starts executing code at the new address. We can see this in assembler using (gdb) **disass foo:**

```
0x1e48 <foo>:    stmw    r30,-8(r1)
0x1e4c <foo+4>:  stwu    r1,-48(r1)
0x1e50 <foo+8>:  mr      r30,r1
0x1e54 <foo+12>: stw     r3,72(r30)
0x1e58 <foo+16>: lwz     r9,72(r30)
0x1e5c <foo+20>: addi    r0,r9,12
0x1e60 <foo+24>: mr      r3,r0
0x1e64 <foo+28>: b       0x1e68 <foo+32>
0x1e68 <foo+32>: lwz     r1,0(r1)
0x1e6c <foo+36>: lmw     r30,-8(r1)
0x1e70 <foo+40>: blr
```

These eleven lines comprise the entirety of the `foo()` function. Once `foo` has finished its work, the `blr`, or *Branch to Link Register*, command tells the processor to jump back to the address stored in the link register. In our case, this is the address `0x1e98`, as stored by `main()` before calling `foo()`.

Calling a function in C is as simple as telling the processor where in memory it needs to jump next. There is an additional layer of indirection added when dealing with functions dynamically linked into your application, but the principle is the same. Calling a function in Objective-C is an entirely different proposition. So much so, that we use a different language and syntax when we talk about it. Instead of calling functions, we *send messages to objects*, and instead of typing `function(arg, ...)` we use `[receiver selector:arg]`.

Under the hood, our fancy Objective-C syntax is turned into an ordinary C function call. You can see this function declared in `/usr/include/objc/objc-runtime.h`:

```
id objc_msgSend(id self, SEL op, ...);
```

`objc_msgSend()` is the cornerstone to Objective-C's dynamism. It is where much of the visible object oriented behavior we use is implemented. Unlike C, or even C++ for that matter, Objective-C waits until run-time to make many decisions. There are two critical questions for message passing that are decided only at run-time by the call to `objc_msgSend()`:

- Which object should be the receiver of the message?
- Is the method associated with the selector defined in the object class, one of its superclasses, or an entirely separate category?

Deciding which object should be the receiver is the easiest part. We make this explicit in selecting a receiver for our messages, which is the first token after the open bracket (`[receiver ...]`). The compiler takes the receiver, and uses it as the first argument to `objc_msgSend()`, named `self`<sup>5</sup>.

Once we know what object is going to receive the message, figuring out which method to invoke seems like it should be relatively simple—but it's not. Let's consider the following class hierarchy:

```
// FooBar.m

@implementation FooBar

...

- (NSString*)description
{
    return [NSString stringWithFormat:@"Foo: %d, Bar: %d", foo, bar];
}

@end

// FooBarToo.h

@interface FooBarToo : FooBar {
}

@end

// FooBarToo.m

@implementation FooBarToo

...

- (NSString*)description
{
    return [NSString stringWithFormat:@"Foo: %d, Bar: %d Too", bar, foo];
}

@end
```

`FooBarToo` inherits from `FooBar`, and both override `NSObject`'s `description` method. Even if we use static typing to reassign a `FooBarToo` object to a `FooBar` pointer, Objective-C's runtime is still smart enough to know we really have a `FooBarToo` on our hands:

```
FooBar *fb = [[FooBar alloc] initWithFoo:12 bar:34];
FooBarToo *fbt = [[FooBarToo alloc] initWithFoo:12 bar:34];

FooBar *fooBar = [fbt retain];

NSLog(@"%@", fb);
NSLog(@"%@", fbt);
```



```
NSLog(@"%@", fooBar);
```

results in:

```
2002-08-27 23:13:21.902 FooBarToo[1207] Foo: 12 Bar: 34
2002-08-27 23:13:21.915 FooBarToo[1207] Foo: 34 Bar: 12 Too
2002-08-27 23:13:21.924 FooBarToo[1207] Foo: 34 Bar: 12 Too
```

This paradigm contrasts with C++, where programmers must explicitly indicate their wish for this behavior by using the `virtual` keyword. Under certain conditions in C++, passing objects by value can even result in sliced objects that lose all but their superclass functionality. C++’s `virtual` keyword makes sure that overridden methods in subclasses are always called, but requires a class developer to explicitly declare methods as `virtual`. In Objective-C *all* methods are automatically “virtual”.

In order for Objective-C to provide true dynamic messaging, where the right method implementation is called every time, the run-time system must be able to figure out to what selectors an object can respond. Let’s take a look at the structure declaration for `objc_class` that we saw in the last section again:

```
struct objc_class {
    struct objc_class *isa;
    struct objc_class *super_class;
    const char *name;
    long version;

    long info;
    long instance_size;
    struct objc_ivar_list *ivars;
    struct objc_method_list **methodLists;
    struct objc_cache *cache;
    struct objc_protocol_list *protocols;
};
```

In addition to the basic information that tells the system about the class, such as its name, version and size, the `objc_class` structure also holds several pointers to structures that allow the Objective-C runtime to pass messages to the correct method implementation. The primary structure that makes this possible is the `methodLists` pointer. `methodLists` is in fact a pointer to an array of `objc_method_list` structures<sup>6</sup>. The Objective-C runtime uses this pointer to an array of structures because it cannot know in advance how many methods a class might have.

Unlike languages such as C++ or Java, Objective-C allows programmers to extend class functionality on-the-fly using *categories*. Categories allow programmers to add methods to a class without subclassing, so they do not exist in the normal chain of inheritance. Each time a category is loaded—whether at application startup, or dynamically when loading a bundle—the Objective-C runtime will add that category’s new methods to the class’ `methodLists` variable as an additional `objc_method_list` structure. That structure is defined as:

```
typedef struct objc_method *Method;

struct objc_method {
    SEL method_name;
```

```

    char *method_types;
    IMP method_imp;
};

struct objc_method_list {
    struct objc_method_list *obsolete;

    int method_count;
#ifdef __alpha__
    int space;
#endif
    struct objc_method method_list[1];    /* variable length structure */
};

```

Every `objc_method_list` structure holds an array of `objc_method` structures, each of which contains three elements: `method_name`, `method_types`, and `method_imp`.

The `method_name` entry in `objc_method` is of type `SEL`, which is defined in `/usr/include/objc/objc.h` as:

```
typedef struct objc_selector  *SEL;
```

Underneath, the selector of a method is defined as a C-string holding the name of the message itself. For example, the selector for `NSObject`'s method `-(BOOL)isKindOfClass:(Class)aClass` is `"isKindOfClass:"`. You are undoubtedly familiar with the Objective-C compiler macro `@selector()`, used to generate `SEL`s from literal method names. For example, if you have an `NSArray` of `NSPortss`, and wanted to invalidate all of the receivers, you could use:

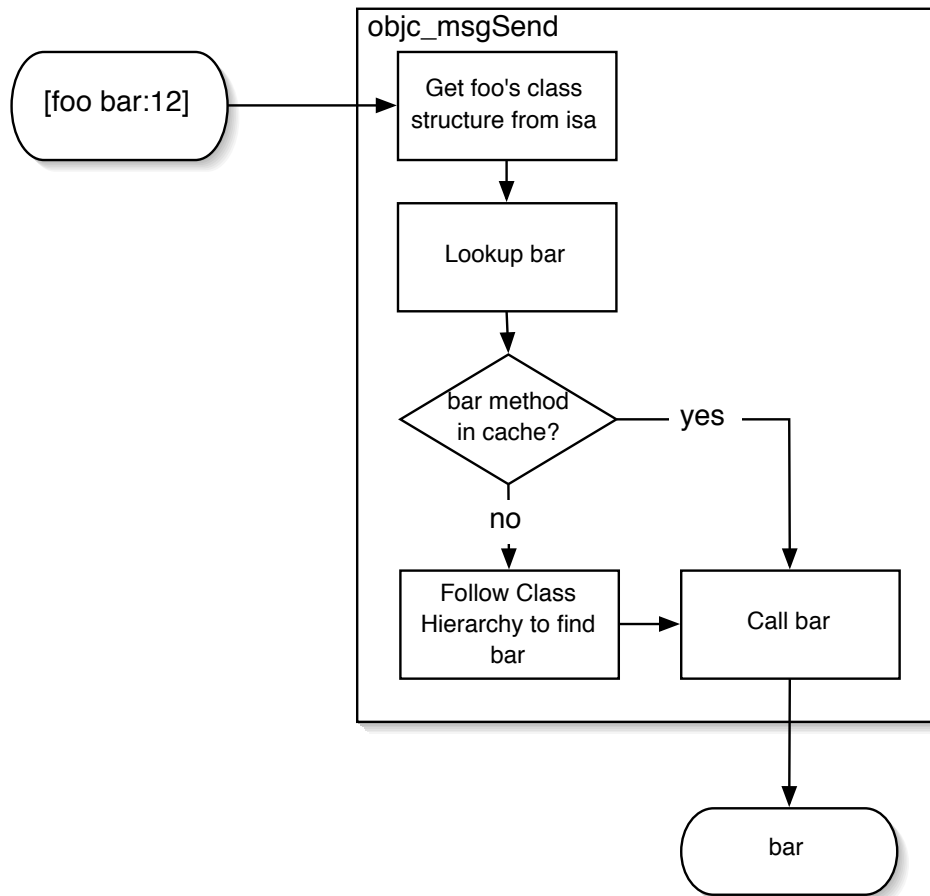
```
[portArray makeObjectsPerformSelector:@selector(invalidate)];
```

Even though `NSArray` knows little about the objects it stores, it does know one critical piece of information: each object should be a subclass of `NSObject`<sup>7</sup>. Knowing this, `NSArray` is able to use `NSObject`'s `performSelector:` method to forward the `invalidate` message.

Once the Objective-C runtime has a selector, it can use the `objc_method` structure to match a method name with its implementation. The `method_imp` entry in `objc_method` stores the actual implementation location of the method. `method_imp` is of type `IMP`, which is defined in `/usr/include/objc/objc.h` as a function pointer returning an `id`:

```
typedef id (*IMP)(id, SEL, ...);
```

The `method_types` variable, like the `ivar_type` variable in `objc_class`, stores information about the types used in the method call. The layout of the C-string is identical to the `IMP` declaration above. That is, first the return type is presented, then the receiving object type—which in our case is always an `id`, then the selector type and finally the arguments. Each of these type values is followed by an offset number. An example entry for `NSObject`'s method `-(BOOL)isKindOfClass:(Class)aClass` would be `c0@4:8#12`. `c` is the `BOOL` return type. `@` represents the type of the receiver, again, always an `id`. `:` is the type of the selector, which is always `SEL`. `#` is the type of the first

Figure 1-7. Sending a Message With `objc_msgSend()`

and only argument: `Class.method_types` uses the same character codes presented in Table 1-1 that are used for determining instance variable type.

Whenever you pass a message to a class, the Objective-C runtime must match the message you wish to send with the method implementation by which it is implemented. This means the runtime needs to search through every `method_list` in an object's `Class` looking for just the right method. `NSObject` alone has 16 `method_list` structures containing a total of 114 methods! This list is generated by Objective-C from the class itself, plus all of its attendant categories.

Luckily, the Objective-C runtime does not have to search the entire `method_list` tree every time it needs to send a message. Instead, the runtime keeps track of all messages it has passed to a class in `objc_class's objc_cache` variable called appropriately: `cache`. The penalty for looking up a method is only incurred the first time it is invoked. Thereafter, `objc_msgSend()` will find the method in the class' method cache. Figure 1-7 provides a diagram of how the `objc_msgSend()` decides which method implementation should receive a message.

### Static Typing Versus id

In Objective-C we can refer to objects as either `id`, or as pointers to their respective types (e.g. `NSString*`). Using these pointers in your code is referred to as *Static Typing*. Static typing can be used by the compiler to check whether you are using your objects properly. For example, the compiler can tell you when you are passing the wrong kind of object as an argument in a selector, or trying to use a selector to which an object does not respond.

There has been some debate in newsgroups and message boards about whether static typing in Objective-C has any impact on *run-time* performance. In every case, the consensus was that the purported boost in speed from using static typing was a placebo effect. Using static typing when writing programs only provides compile-time type safety.

Why *couldn't* **gcc** use static typing to shortcut the run-time system's dynamic dispatch by linking straight to the method implementation? Unfortunately, this would break the expected behavior of inheritance. While we might declare an object to be of a static type—say `NSButton*`, what we are really saying is the object is of type `NSButton`, or *any of its subclasses*. It is not until runtime that we may definitively know which actual instance method will get called.

#### 1.4.2.1. Key-Value Coding

Objective-C has been described as a verbose language. It does not shy away from long lines of descriptive code. What it lacks in brevity, it makes up for in flexibility. One area where Objective-C's open object implementation is used to good effect is in the `NSKeyValueCoding` protocol:

```
@interface NSObject (NSKeyValueCoding)

- (id)valueForKey:(NSString *)key;
- (void)takeValue:(id)value forKey:(NSString *)key;
- (id)storedValueForKey:(NSString *)key;
- (void)takeStoredValue:(id)value forKey:(NSString *)key;
+ (BOOL)accessInstanceVariablesDirectly;
+ (BOOL)useStoredAccessor;

@end
```

All classes derived from `NSObject` automatically inherit an implementation of the `NSKeyValueCoding` protocol, which allows programmers to access the instance variables of a class using a string-based key. You don't have to know in advance what all of a class' instance variable's names might be. the `-valueForKey:` and `-takeValue:forKey:` methods are able to take a string holding the name of a class' instance variable, and get or set that variable's value.

The default implementation actually tries several ways at pulling this off:

1. Given a key named `key`, look for accessor methods named `-setKey:`, `-key` or `-getKey`.
2. If any of those methods are not available, the implementation will look for the above method names with leading underscores: `-_setKey:`, `-_key` or `-_getKey`

3. If there are no accessor methods, the implementation will look directly for instance variables named either `key` or `_key`.
4. If none of the above work, the `NSKeyValueCoding` implementation will give the class a chance to handle the unbound key, or throw an exception.

The `NSKeyValueCoding` implementation is able to accomplish all of this using the information made available through the `objc_class` structure and supporting functions.

#### 1.4.2.2. Optimizing with `IMPS`

Even with caching, the Objective-C runtime must still perform a lookup each time a message needs to be sent to an object. In most situations, this added level of indirection is negligible. However, there are certain situations in which you might want to bypass Objective-C's dynamic dispatch for a faster, static approach. Consider the following method from Lignarius's `RRGGOptimizer` class:

```
- (void)optimize
{
    BOOL finished = NO;

    // zero out the structures
    [rectangleValues removeAllObjects];
    [verticalCuts removeAllObjects];
    [horizontalCuts removeAllObjects];
    [self setValue:0 forSize:stockSize];

    first.width = wStepping;
    first.height = hStepping;
    second.width = wStepping;
    second.height = hStepping;

    [self findVerticalCut];
    [self findHorizontalCut];

    while(!finished) {
        while(second.width < stockSize.width) {
            second.width += wStepping;
            [self findVerticalCut];
            [self findHorizontalCut];
        }
        if(second.height < stockSize.height) {
            second.height += hStepping;
            second.width = wStepping;
            [self findVerticalCut];
            [self findHorizontalCut];
        } else {
            finished = YES;
        }
    }
}
```

When looking for an optimal layout for a given set of parts and stock material, this is the method that kicks it all off. You may notice that there are a couple Objective-C messages called within the main loop that comprise the bulk of this method's computation.

`-findVerticalCut` and `-findHorizontalCut` are deep in the nested loops, and depending on the size of the stock cutting problem, selectors could be sent thousands of times. Each time, the Objective-C runtime system must follow the steps outlined in Figure 1-7 to find the proper method implementation. An important optimization we can make is to perform this implementation lookup outside of our loop beforehand. By caching the location of the implementation, we can call the implementation directly, thus bypassing the costly lookup every time through the loop.

Recall that the definition of `IMP` is:

```
typedef id (*IMP)(id, SEL, ...);
```

In other words, a pointer to a function returning `id`. While we could trudge through the `objc_class` structure in order to find a method's associated `IMP`, `NSObject` provides a handy method for obtaining the `IMP` from a selector. Given an Objective-C class instance method returning `id`, we would grab a method's implementation by calling:

```
- (IMP)methodForSelector:(SEL)aSelector
```

Once we have a method's implementation pointer, we can call it like any traditional C function pointer. Consider the following method from `NSDictionary`:

```
- (id)objectForKey:(id)key;
```

We could call this method using its `IMP` as follows:

```
IMP objectForKeyIMP =
    [dict methodForSelector:@selector(@"objectForKey:")];

while(...) {
    object = objectForKeyIMP(dict, @selector(@"objectForKey:"), key);
    ...
}
```

Unfortunately, not all methods match the `IMP` typedef. In the case of `findVerticalCut` and `findHorizontalCut` our method returns `void`. In these cases, you must make your own function pointer, and cast the return of `methodForSelector` like:

```
void (*findVerticalCutIMP)(id, SEL);
findVerticalCutIMP = (void (*)(id, SEL))[self
```

```
methodForSelector:@selector(@"findVerticalCut")];
```

Our optimize method, without the use of Objective-C messaging, now looks like:

```
- (void)optimize
{
    // ...
    void (*findVerticalCutIMP)(id, SEL);
    void (*findHorizontalCutIMP)(id, SEL);

    findVerticalCutIMP = (void (*)(id, SEL))[self
        methodForSelector:@selector(@"findVerticalCut")];

    findHorizontalCutIMP = (void (*)(id, SEL))[self
        methodForSelector:@selector(@"findVerticalCut")];

    findVerticalCutIMP(self, @selector(@"findVerticalCut"));
    findHorizontalCutIMP(self, @selector(@"findHorizontalCut"));

    while(!finished) {
        while(second.width < stockSize.width) {
            second.width += wStepping;
            findVerticalCutIMP(self, @selector(@"findVerticalCut"));
            findHorizontalCutIMP(self, @selector(@"findHorizontalCut"));
        }
        if(second.height < stockSize.height) {
            second.height += hStepping;
            second.width = wStepping;
            findVerticalCutIMP(self, @selector(@"findVerticalCut"));
            findHorizontalCutIMP(self, @selector(@"findHorizontalCut"));
        } else {
            finished = YES;
        }
    }
}
```

We have now eliminated any Objective-C message passing from the main loop of the `optimize` routine. This does not, however, guarantee that there are no messages being sent from within the `findVerticalCut` and `findHorizontalCut` methods themselves.

This technique is obviously not necessary every time we need to send a message to an Objective-C object. What are the right times to use it then? If the following two conditions are met, your implementation may benefit from this optimization:

1. You are sending many messages to an object in a tight loop.
2. The object itself is not changing in a way that would alter the location to which the IMP was pointing.

Given the above, it is important to remember that you need to be sending many hundreds or thousands of messages in repetition to an object for this technique to make a noticeable difference.

This technique will also not work if there is a chance for the `IMP` to change. Imagine a situation where you have a list of `Shape` objects to which you want to send the message `-area`. If those objects are all each actually subclasses of type `Rectangle`, `Circle`, etc., that override the `-area` method, you won't know beforehand which method implementation you will need to call for any particular object. `Rectangle` objects will want to use their `-area` implementation and `Circle` objects will want to use their's. Be wary if you are only holding pointers to superclass objects and wanting to use the `IMP` invocation optimization.

### 1.4.3. Destroying Objects

We have talked about how objects are creating in Objective-C, as well as how they are used. The final stage in an object's lifecycle is its destruction. The act of destruction itself is not that difficult to understand—when you are done with an object, remove its instance from memory. It is determining when you are done with an object that turns out to be a more than trivial task. Objective-C, through the `NSObject` class solves the problem of knowing when to send objects to the great bit-bucket in the sky using a technique known as reference counting.

In traditional C programming, dynamic memory allocation using `malloc()` must be paired with an equal number of `free()`s. This memory management pattern, while conceptually simple, is in practice not easy to get right. This is especially true when calling system-level routines or third party libraries where you may not have the original source code. If you are handed a block of memory through a pointer, is it your responsibility to free it, or will the system? What if you then pass it on to a third library, is that library going to release it out from under you? Ultimately, the `malloc/free` pattern does not address the issue of memory lifetime and ownership itself. It is only through careful coordination between software developers that leaks can be avoided.

This confusion over memory "ownership" is especially confusing when dealing with encapsulated objects. Different object oriented languages have tried dealing with this problem in many ways. The two most popular methods for managing memory are reference counting (RC) and garbage collection (GC). Of the two methods, RC is conceptually the simplest. When you are given an object that you want to hold beyond the current scope, you increment a *reference count* variable associated with the object. When you are finished with the object you decrement the reference count. If the count falls to zero, the system knows no one else is interested in the object, and releases it.

GC has been around for a long time, but was popularized by the Java programming language. GC starts with the same principle as RC, but automates the process. Reference counts are automatically incremented and decremented by the runtime as objects come into and go out of scope. When an object's reference count falls to zero, the object is automatically destroyed. GC allows programmers to by and large avoid issues of memory management. Java's GC implementation is a significant part of what makes Java such an easy language for new programmers. The biggest complaint leveled against GC is that it can be slow. The overhead of keeping track of these references in the runtime system imposes a penalty on the entire application.

#### 1.4.3.1. Objective-C Reference Counting

The Objective-C language itself doesn't specify a memory management technique inasmuch as it leaves this detail to library developers. The root class for all objects in the Foundation and AppKit libraries, `NSObject`, uses reference counting. You are probably already familiar with the rules for memory management in Cocoa. You are only responsible for releasing an object if you have done one of the following:

1. allocated and initialized an object
2. retained an object
3. copied an object



You are not responsible for releasing any other objects, no matter how you might have come to hold a reference to that object.

### 1.4.3.2. Autorelease Pools

While reference counting alone works well, autorelease pools provide an additional level of automation in memory management by automatically releasing objects when their reference count falls to zero. Autorelease pools can be viewed as an attempt at giving reference counting some of the ease-of-use of garbage collecting.

By sending `-autorelease` to an object, you place it in the current autorelease pool. At the end of the current event loop, or when the autorelease pool itself is released, it goes through the list of its objects and checks to see if any object's reference count is zero. If it finds an object with no references, it sends the `-release` message to that object. Just as with garbage collection, there is an additional overhead when using autorelease pools in your code. judicious use of autorelease pools can be an important optimization technique.

Reference counting, autorelease pools and general memory management in Objective-C will be covered in much more detail in .

## 1.5. Back Together Again: Objective-C++

Mixing C and Objective-C code has never been a problem on OS X. In fact, some of the functionality provided by Cocoa is simply a wrapper around C-based Carbon libraries. This means you can mix and match C code as appropriate in your applications. There are plenty of reasons you might want to fall back on C. Most likely, it will be because you either have legacy code with which you need to inter-operate, or find the need to implement highly optimized routines. Whatever the reason, for all practical purposes, Objective-C *is* C.

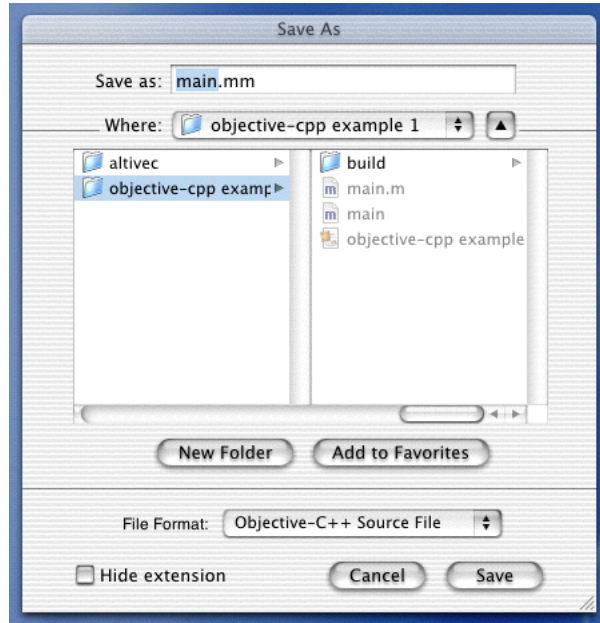
The situation is not as simple with C++. Previously on OS X, if you wanted to mix Objective-C and C++ you could only use their common denominator, C. This typically meant programmers had to develop compatibility layers that necessarily strip the object oriented nature of either the Objective-C or C++ libraries, thus rendering much of their attraction moot. It turns out that NeXT had a version of Objective-C that could inter-operate with C++ code called (not surprisingly) Objective-C++, that at first did not make NeXTStep's transition to the Macintosh platform. Apple soon rectified the situation, and later OS X developer tools include support for the Objective-C++ language. Objective-C++ is not widely advertised by Apple, but it exists, and has been updated to support the current C++ standard.

Support for Objective-C++ is built into **gcc** and ProjectBuilder. ProjectBuilder uses the file extension to determine in which language a file has been written. For Objective-C++, the expected extension is `*.mm`. If you choose **File**→**Save As...** in ProjectBuilder you have the option of selecting an Objective-C++ file type (See Figure 1-8).

To get an idea of what Objective-C++ can do, let's look at an example. The following code is the header file for a very simple C++ class called `Widget`:

```
#include <string>

class Widget {
    friend ostream& operator<<(ostream &os, const Widget &w);
```

**Figure 1-8. ProjectBuilder File Type for Objective-C++**

```
public:
    string getName();
    void setName(string new_name);

protected:
    string name;
};
```

Notice that the `Widget` class declaration uses the standard

```
#include <string>
```

syntax for include files, as well as the other traditional fare you would expect for a C++ class. Objective-C++'s compliance with the C++ standard follows from **gcc**'s support of the standard. Earlier versions of **gcc** (pre 3.x) had some limitations in this regard. The current C++ support in **gcc** is much more complete. `Widget` is obviously a pretty simple beast, so we probably won't be stressing the C++ implementation too any great degree. It has one member variable: `name`, two methods: `getName` and `setName` as well as a single friend function: the overloaded left-shift operator (`<<`).

The implementation file `widget.cpp` for `Widget` is:

```
#include "widget.h"

string Widget::getName()
```

```

{
    return name;
}

void Widget::setName(string new_name)
{
    name = new_name;
}

ostream& operator<<(ostream &os, const Widget &w)
{
    return os << w.name;
}

```

Again, the `Widget` class implementation follows the standard conventions for C++ programming. `Widget` is a full fledged C++ class in all respects.

Using this C++ class in our Objective-C++ code turns out to be very easy. The following listing is the `main.mm` file for a very simple application showing how Objective-C++ can mix Objective-C and C++ code.

```

#import <Foundation/Foundation.h>
#import "widget.h"

int main (int argc, const char * argv[])
{
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
    Widget w;
    NSString* s = @"Widget Number 1";

    w.setName([s cString]);

    NSLog([NSString stringWithCString:w.getName().c_str()]);
    cout << w << endl;

    [pool release]; return 0;
}

```

Like all Objective-C programs, we start by creating an autorelease pool to take care of any Objective-C objects we might create along the way. Our application then creates two objects—a `Widget w` and an `NSString s`. Next, we set `w`'s name using `NSString`'s `cString` selector, which returns a traditional null-terminated `char*`. Since `Widget`'s `setName()` method expects a `string`, `string`'s C++ style constructor is implicitly called. We then get `w` to output its name using both the Foundation framework's `NSLog()` function, and C++'s `cout` object. In the case of `NSLog()`, we have to get a C-style string from `w`, while `cout` uses our overloaded friend `<<`. Finally, we follow proper Objective-C convention and release our pool, and exit out of `main`.

Despite this example's small amount of code, there is a lot going on under the hood. If you take a second to consider what we have just done, it is actually pretty amazing. Without very much fuss, we have integrated objects from two different languages. There are a couple features of Objective-C++ that make this transition easy: First,

Objective-C++ allows us to program with the full range of both Objective-C and C++ syntax. We are not limited to a poor-man's subset of commonality. If you want to use the full expressiveness of C++'s operator overloading, feel free to do so. If you want to incorporate Distributed Objects or Objective-C's dynamism, do so as well. Second, while our use of the `Widget` class shows how we can incorporate custom C++ objects, this code also shows how we have all of the standard C++ libraries at our disposal as well. We were able to implicitly and explicitly use the `string` class and `cout` object.

If you need to work with existing C++ libraries, ProjectBuilder's Objective-C++ implementation makes your work fairly painless. There is no need to create a bridging layer of code between your Objective-C and C++ work. In addition, there are a few side benefits to using ProjectBuilder's Objective-C++ mode. In addition to using C++'s object-oriented features, you can also take advantage of the more mundane changes C++ makes to C syntax. Take the following Objective-C implementation as an example:

```
#import "widget.h"

@implementation Widget

- (void)doSomething {
    NSString *s = @"foo";
    NSLog(s);

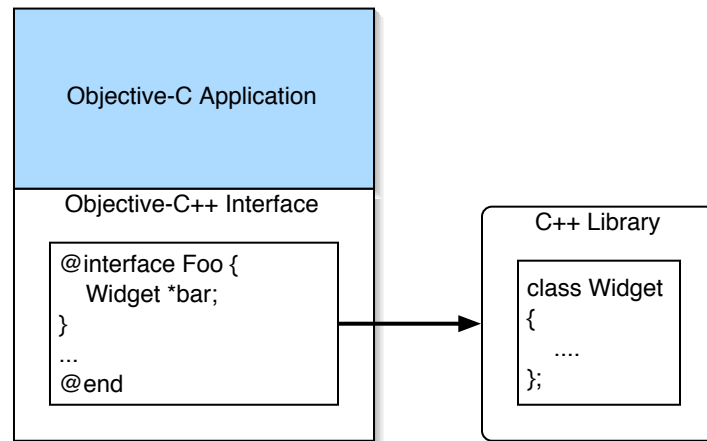
    for(int i = 0; i < 10; ++i) {
        // do something clever
    }
}

@end
```

Save it as `widget.m`, and **gcc** will give you: `widget.m: In function '-[Widget doSomething]': widget.m:10: parse error before 'int'`

However, save the exact same file as `widget.mm`—an Objective-C++ file type, and **gcc** will happily compile it. In this case, Objective-C is using the old C rules about where we can declare variables in a block of code, which means that all of our variables must be declared at the *beginning* of the block. As Objective-C++ code, we get to declare variables anywhere we wish, including within the `for` loop expression. Some programmers have even been known to automatically use the `*.mm` file extension in order to simply avoid the more restrictive syntax of C!

Apart from avoiding C limitations, there are actually more productive reasons for using Objective-C++. Typically, this is to integrate C++ and Objective-C code into a single application. There are many ways that a programmer might want to combine Objective-C and C++; however, for the sake of simplicity, I would recommend keeping a safe distance between the Objective-C and C++ portions of your code. It is possible to turn otherwise perfectly well behaved Objective-C classes in hybrid monsters that devour your productivity. Just because Objective-C++ lets you freely intermingle C++ and Objective-C code is no reason to abandon good design choices. Good application design will isolate libraries behind well documented interfaces, so that as implementations change, an entire application does not have to be retooled. Objective-C++ provides the bridge with which to create this interface.

**Figure 1-9. Calling C++ from Objective-C**

### 1.5.1. Using C++ Classes from Objective-C

The most common use of Objective-C++ is probably when developers need to integrate existing C++ libraries into their code. Figure 1-9 depicts this relationship.

Take another look at the `main.mm` file from the `Widget` example:

```
#import <Foundation/Foundation.h>
#import "widget.h"

int main (int argc, const char * argv[])
{
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];


    Widget w;
    NSString* s = @"Widget Number 1";

    w.setName([s cString]);

    NSLog([NSString stringWithCString:w.getName().c_str()]);
    cout << w << endl;

    [pool release];
    return 0;
}
```

Now, what would you consider this application—an Objective-C program using C++ objects, or a C++ program using Objective-C objects? In fact, you could pick either position and be correct. This example is a little *too* simple, in that no Objective-C or C++ object directly interacts with the other language's constructs. If you look closely, we are simply passing around pointers to characters between the objects. In any real Objective-C application, `main()`

**Figure 1-10. A Simple Table View**


Index	Wood	Board Feet
0	Cupertino	Cupertino
1	San Jose	San Jose
2	Santa Clara	Santa Clara
3	San Francisco	San Francisco
4	Palo Alto	Palo Alto
5	San Carlos	San Carlos
6	Los Gatos	Los Gatos
7	Sunnyvale	Sunnyvale
8	Mountain View	Mountain View
9	Redwood City	Redwood City

will be a distant memory up the calling stack, and we will need to interact with our C++ libraries from within proper Objective-C objects.

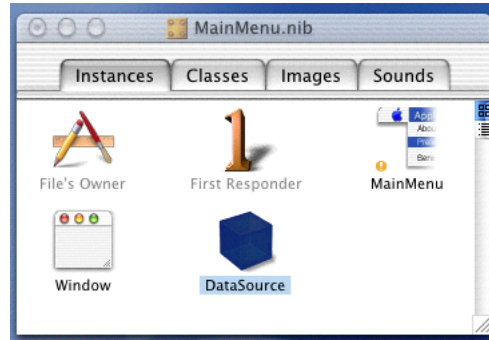
First, we need to properly isolate the portions of our code that will interact with C++ from the portions of our code that will be pure Objective-C. The best way to do this is through the use of *protocols*. Here is our simple protocol definition:

```
@protocol DataSourceProtocol <NSObject>
- (void)loadFilename:(NSString*)filename;
@end
```

There are a few things to remember. First, the table view datasource is the Objective-C++ object. notice that the table view doesn't care. It happily sends messages to the Objective-C++ datasource. This is the advantage of the dynamic nature of Objective-C. Even though tableview is a pure Objective-C object, we can set its datasource outlet to our Objective-C++ object, and it happily trundles away. However, if you were to try to directly set the table view's datasource to the Objective-C++ object you would have difficulty getting things to compile. The use of the protocol helps to better define our intentions to the compiler. Also, We cannot include the datasource header file directly since it has C++ portions that will fail to compile in combination with the Objective-C-only portions of the code as we will see next.

Here is the header file for our Objective-C++ object. In this case, we are interested in using C++'s collection classes from the STL. The DataSource object is going to hold a pointer to a vector storing the information our program is interested in.

```
#import <Cocoa/Cocoa.h>
```

**Figure 1-11. Create an Instance of our Objective-C++ Object DataSource**

```
#include <vector>
#include <string>

#import "DataSourceProtocol.h"

@interface DataSource : NSObject <DataSourceProtocol> {
    vector< pair<string, float> > *data;
}
@end
```

At this point, we can instantiate it in our nib, and connect it to the table view's datasource (See Figure 1-11).

The datasource class needs to implement a couple of protocols, both formal and informal, that are not apparent in its header file. The first of these is the tableview data source *informal* protocol. You need to accept two messages: **numberOfRowsInTableView:** and **tableView:objectValueForTableColumn:row:**. In addition, datasource implements the formal protocol we declared in **DataSourceProtocol.h**, which is the **loadFilename:** message.

```
#import "DataSource.h"
#include <iostream>
#include <fstream>

@implementation DataSource

- (id)init
{
    if(self = [super init]) {
        data = new vector< pair< string, float> >;
    }
    return self;
}

- (void)loadFilename:(NSString*)filename
{
```

```

ifstream infile([filename cString]);
string wood;
float feet;

if(!infile) {
    NSRunAlertPanel(@"Error",
        [NSString stringWithFormat:@"Error Opening File %@", filename],
        nil, nil, nil);
    return;
}

while( infile >> wood ) {
    infile >> feet;
    data->push_back( make_pair< string, float > (wood, feet));
}

- (int)numberOfRowsInTableView:(NSTableView *)aTableView {
    return data->size();
}

- (id)tableView:(NSTableView *)aTableView
objectValueForTableColumn:(NSTableColumn *)aTableColumn row:(int)rowIndex
{
    if([[aTableColumn identifier] isEqualToString:@"index"])
        return [NSString stringWithFormat:@"%d", rowIndex];

    if([[aTableColumn identifier] isEqualToString:@"wood"])
        return [NSString stringWithCString:((*data)[rowIndex]).first.c_str()];

    if([[aTableColumn identifier] isEqualToString:@"feet"])
        return [NSString stringWithFormat:@"%f", ((*data)[rowIndex]).second];

    return @" ";
}

- (void)dealloc {
    delete data;
    [super dealloc];
}

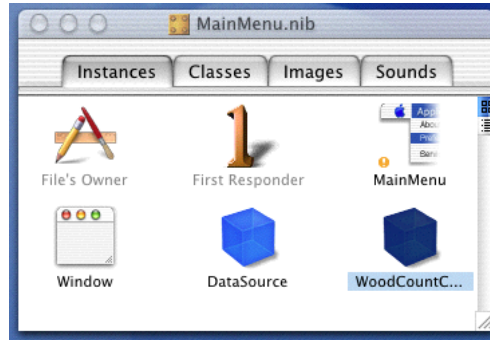
@end

```

Notice that in our `-init` method we dynamically create the C++ vector, and properly dispose of it in our `-dealloc` method. This is a very important point:

*Hold all objects as pointers.* Just like your statically typed Objective-C objects, make sure you use pointers to C++ objects as instance variables, instead of the objects themselves. Since the Objective-C run-time isn't familiar with the C++'s object model, it will not be able to properly instantiate the variables. It will not automatically call the C++ constructors and destructors as you are familiar with. By using pointers, you can properly construct the instance variables in your `-init`



**Figure 1-12. Create an Instance of the WoodCountController Class**

method, as well as properly destroy the variable in your `-dealloc` method. The same goes for Objective-C classes as members of C++ objects.

We now have a completed datasource object that bridges the gap between the C++ vector and pair classes and Objective-C. It does so without undue exposure to C++ in the rest of the application.

There is an additional class we need to complete our application. That is a controller to load the `NSOpenPanel`, and tell the datasource to load the new data. It uses the formal protocol we defined earlier to do this. Here is the header file:

```
#import <Cocoa/Cocoa.h>
#import "DataSourceProtocol.h"

@interface WoodCountController : NSObject {
    IBOutlet NSTableView* table;
    IBOutlet id<DataSourceProtocol> datasource;
}
- (IBAction)open:(id)sender;
@end
```

as with the datasource class, create an instance of the `WoodCountController` class in Interface Builder (See Figure 1-12).

```
#import "WoodCountController.h"

@implementation WoodCountController

- (IBAction)open:(id)sender
{
    NSOpenPanel *panel = [NSOpenPanel openPanel];
    [panel setDelegate:self];
    [panel setPrompt:@"Open"];
}
```

**Figure 1-13. Our New Mixed Objective-C and C++ Application Running**


Index	Wood	Board Feet
0	Ash	24.10
1	Cherry	43.80
2	Maple	100.00
3	Oak	265.00
4	Pine	300.00
5	Walnut	86.00
6	Mahogany	10.00
7	Teak	9.00
8	Balsa	18.00

```

[panel beginSheetForDirectory:nil file:nil types:nil
    modalForWindow:[table window] modalDelegate:self
    didEndSelector:@selector(openPanelDidEnd:returnCode:contextInfo:)
    contextInfo:[NSString stringWithString:@"open"]];
}

- (void)openPanelDidEnd:(NSOpenPanel*)openPanel
returnCode:(int)returnCode contextInfo:(void*)x
{
    if(returnCode == NSOKButton) {
        [datasource loadFilename:[openPanel filename]];
        [table reloadData];
    }
}

@end

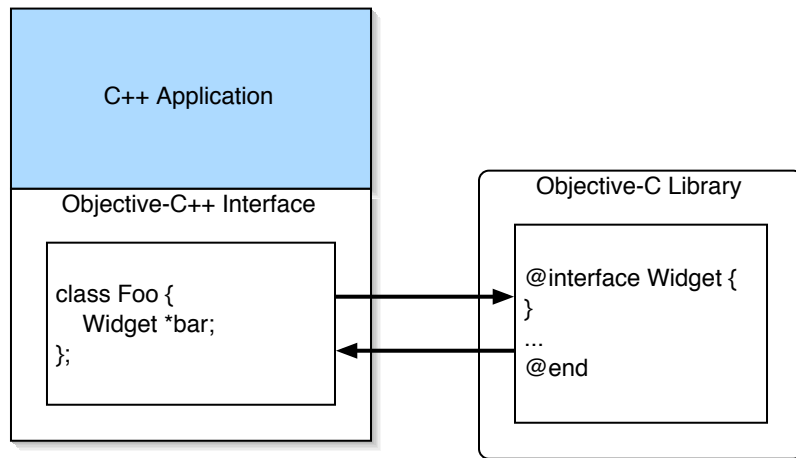
```

### 1.5.2. Using Objective-C Classes from C++

Most programmers used Objective-C in the past in order to program for NeXTStep, and the same holds true for OS X<sup>8</sup>. When you find yourself needing Objective-C++, 99.9% of the time you will probably be writing a graphical application for OS X, and needing to integrate your favorite C++ library. A good architecture would dictate proper separation between the two languages by a strong interface. Because of modern GUIs' use of event-driven design, control typically moves through an application from the visible elements, written in Objective-C, to the back-end, where the majority of the C++ code exists. This means that Objective-C code will call the C++ code more often than the other way around.

In what cases might the situation be reversed?

1. Your application might in fact be written using a mix of C and C++ and the Carbon libraries, and you want to take advantage of something in Cocoa.

**Figure 1-14. Integrating Objective-C into a C++ Program**

2. Algorithms or routines in your C++ code need access to data structures best kept in the Objective-C realm.

In these cases, it might make more sense to flip our point of view (see Figure 1-14). When we were integrating C++ into our Objective-C application, we created a bridge by creating Objective-C classes in our interface. For integrating Objective-C code into a C++ program, we will do the reverse. Our bridging code will be written as a C++ class holding pointers to the Objective-C objects we wish to incorporate.

From C++, Objective-C objects look like exactly what they are: pointers. In other words, you don't change your normal Objective-C object syntax when referring to Objective-C objects in C++.

We cheated with the previous example by using Interface Builder to side-step the creation of the Objective-C++ object. In this case, we are again avoiding intermingling our C++ code with Objective-C code, except we are using a factory method with which to accomplish this.

```

#import <Foundation/Foundation.h>
#import "LServerProtocol.h"
#import "LClientProtocol.h"

@interface LClient : NSObject <LClientProtocol> {
    id server;
    id delegate;
}
- (id)delegate;
- (void)setDelegate:(id)del;
- (void)registerWithServerName:(NSString*)serverName hostName:(NSString*)hostName;
- (NSDictionary*)getCutList;
- (void)completedWork:(NSDictionary*)cutList;
@end

@interface LClientDelegateProtocol : NSObject {

```

```

}
- (void)serverTerminatingForClient:(LClient*)lClient;
@end

#import "LClient.h"

static int runs = 0;

@implementation LClient

- (id)delegate
{
    return delegate;
}

- (void)setDelegate:(id)del
{
    [delegate autorelease];
    delegate = del;
}

- (void)registerWithServerName:(NSString*)serverName
hostName:(NSString*)hostName
{
    server = [NSConnection rootProxyForConnectionWithRegisteredName:serverName
                                                host:hostName];
    [server setProtocolForProxy:@protocol(LServerProtocol)];

    [server registerClient:self];
}

// ClientProtocol Implementation

- (void)serverTerminating
{
    NSLog(@"terminating");

    if(delegate)
        [delegate serverTerminatingForClient:self];
}

// methods encapsulating the server object

- (NSDictionary*)getCutList
{
    NSLog(@"getting cut list");

    if(++runs > 5)
        [self serverTerminating];
}

```

```

    return [server getCutList];
}

- (void)completedWork:(NSDictionary*)cutList
{
    NSLog(@"sending");
    [server completedWork:cutList];
}

@end

#include <utility>
#include <string>
#include <vector>

typedef pair<float, float> rectangle;

class LignariusClientProtocol {
public:
    virtual ~LignariusClientProtocol() {}
    virtual void registerWithServer(const string &serverName,
                                    const string &hostName) = 0;
    virtual vector<rectangle>& getCutList() = 0;
    virtual void completedWork(vector<rectangle>& cl) = 0;

    bool terminated;
};

class LignariusClientFactory {
public:
    static LignariusClientProtocol* newLignariusClient();
};

#include "LignariusClient.h"

LignariusClientProtocol* LignariusClientFactory::newLignariusClient()
{
    return (new LignariusClient);
}

```

Notice in Figure 1-14 that interaction between the C++ and Objective-C objects goes both directions: as with many Objective-C objects, there is a delegate. In our case, we want to feed this delegate information back to C++ objects. We need to feed an Objective-C object to LClient, but have that object call back into a C++ object. So, our Objective-C++ interface has two classes declared. One in C++, and one in Objective-C.

```

#include <Foundation/Foundation.h>
#include "LignariusClientProtocol.h"
#import "LClient.h"

class LignariusClient;

@interface LignariusClientCallback : NSObject {
    LignariusClient *callback;
    LClient *client;
}
- (id)initWithCallback:(LignariusClient*)cb lClient:(LClient*)lc;
@end

class LignariusClient : public LignariusClientProtocol {
public:
    LignariusClient();
    virtual ~LignariusClient();
    void registerWithServer(const string &serverName, const string &hostName);
    vector<rectangle>& getCutList();
    void completedWork(vector<rectangle> &cl);
    void serverTerminated(bool term);

private:
    LClient *client;
    LignariusClientCallback *cbProxy;
    vector<rectangle> rectangles;
};

```

Here is the implementation files for both the Objective-C and C++ classes:

```

#import <Foundation/Foundation.h>
#include "LignariusClient.h"

@implementation LignariusClientCallback

- (id)initWithCallback:(LignariusClient*)cb lClient:(LClient*)lc
{
    if(self = [super init]) {
        callback = cb;
        client = lc;
        [client setDelegate:self];
    }
    return self;
}

- (void)serverTerminatingForClient:(LClient*)lClient
{
    if(client = lClient)
        callback->serverTerminated(true);
}

```

```

@end

LignariusClient::LignariusClient()
{
    client = [[LClient alloc] init];
    cbProxy = [[LignariusClientCallback alloc] initWithCallback:this
                lClient:client];
}

LignariusClient::~~LignariusClient()
{
    [client release];
    [cbProxy release];
}

void LignariusClient::registerWithServer(const string &serverName,
                                         const string &hostName)
{
    [client registerWithServerName:[NSString
    stringWithCString:serverName.c_str()]
    hostName:[NSString
    stringWithCString:hostName.c_str()]];
}

vector<rectangle>& LignariusClient::getCutList()
{
    // ...
    [client getCutList];
    return rectangles;
}

void LignariusClient::completedWork(vector<rectangle> &cl)
{
    // ...
    NSDictionary* dict = [[NSDictionary alloc] init];
    [client completedWork:dict];
}

void LignariusClient::serverTerminated(bool term)
{
    terminated = term;
}

```

As with handling C++ objects from Objective-C, the C++ object model does not automatically take into account the reference counting and autorelease pools of Objective-C. The C++ runtime will not automatically call Objective-C constructors or destructors. When you are finished with an Objective-C object, make sure you properly release it if necessary.

Finally, we can see the pure C++ `main()` implementation.

```

#include <iostream>
#include "LignariusClientProtocol.h"

int main (int argc, const char * argv[])
{
    // insert code here...
    cout << "Starting Client...\n";

    LignariusClientProtocol* c = LignariusClientFactory::newLignariusClient();

    while(!c->terminated) {
        vector<rectangle> cl = c->getCutList();

        // do our work // ...

        c->completedWork(cl);
    }
    cout << "Completed\n";
    return 0;
}

```

### 1.5.3. Things to Watch Out For

There are several limitations to keep in mind when using Objective-C++. These limitations arise from the fact that when a program incorporating Objective-C and C++ code is run, the runtime systems that support each language are entirely separate, and therefore do not know how to handle each other's objects. For example, you cannot use Objective-C messaging syntax on C++ objects, or intermingle C++ and Objective-C class hierarchies. Check the release notes for a full list of specific language limitations (see Section 1.6). In most cases, if you follow the design suggestion of properly insulating your Objective-C and C++ code from each other, you can avoid many of the problems; however, there are still a couple of issues with using Objective-C++ you may want to consider.

Using the different exception systems from C++ and Objective-C can be a little tricky. Objective-C's `NSException` class uses the `setjmp`, `longjmp` calls, while C++'s exception system is built into the language and run-time support. Mixing Objective-C and C++ code that might throw exceptions takes a little extra effort.

Like almost all compilers on the market, support for the full C++ standard in **gcc** is not always what one would hope for. The C++ standard, including the STL, is enormous, and it takes time to implement every feature, and even when a feature is implemented, it may not be stable. The latest ProjectBuilder release includes **gcc** version 3.1, which has much better support for the C++ standard ??? note: check before book goes out to get latest updates ??? Particularly troublesome for the compiler and library vendors has been full support for generic programming and templates. Many advanced C++ computational libraries make use of more subtle template features, and test compiler implementations to their breaking point. As a rule of thumb, if the C++ code you are interested in using works with **gcc**, you should be okay.

Both Objective-C and C++ impose an overhead on applications over straight C. When you combine the two, you get the overhead of both. What does this mean in practical terms? The first hit you will take is in compile speed. **gcc** is not fast at the best of times, so you can expect its speed on the Objective-C++ portions of your code to be slower than C or Objective-C alone. **gcc** is always getting better in this regard, and there are ways to speed up your compile times.



NOTE: Currently, ProjectBuilder does not by default precompile headers for Objective-C++ code. This can lead to long compile times when including C++ code in your projects. In order to turn on pre-compiled headers, use the `-cpp-precomp` option. Newer versions of ProjectBuilder include **GCC 3.1**, or newer, and support an entirely new pre-compiled header system. ??? Note: check this before book goes out to get the latest release info on dev tools ???

The second hit you will take with Objective-C++ is in application size. However, this is a relatively small penalty. For a minimal program, compiling the same code as Objective-C and again as Objective-C++ results in a 4KB difference. If you have **gcc** generate debugging symbols, the Objective-C++ code will be much larger in order to include the extra information. Application file size is only one half of the equation. You will also see an increase in the runtime memory requirements of your application as it will require the C++ libraries in order to run. As long as you do not statically link the libraries, this cost is amortized over all of the C++ applications running the shared library.

Luckily, there is really no impact on the runtime speed of your application after you have compiled your code as Objective-C++ instead of Objective-C. One of C++'s goals has been to avoid imposing overhead for features programmers are not using. This is one reason garbage collection has been avoided in the standard for such a long time. If you are not using a feature of C++, then you should not see an impact on the runtime performance of your application, apart from the pressure loading larger libraries creates on the memory subsystem. In general, this will only be apparent during program loading, and low memory situations, where the system is forced to swap pages to and from disk to support your application.

With many high performance libraries already in existence for C and C++, the ability to seamlessly integrate this code into Cocoa applications can save you the effort of reinventing the wheel. If you need to use C++, Objective-C++ is a great tool to have around. Apple's developer tools let you incorporate mixed C, C++ and Objective-C code all in one application. For this reason, don't think of Objective-C++ as a different language, but more of a C++ compatibility switch you can throw in Objective-C when needed.

## 1.6. Resources

### Resources

#### Mach-O Runtime Architecture Document

<http://developer.apple.com/techpubs/macosx/DeveloperTools/MachORuntime/MachORuntime.pdf>

#### class-dump

[http://www.omnigroup.com/\\$\sim\\$nygard/Projects/](http://www.omnigroup.com/$\sim$nygard/Projects/)

#### F-Script

<http://www.fscript.org/>

#### Objective-C++ Release Notes

<file:///Developer/Documentation/ReleaseNotes/Objective-C++.html>

## Notes

1. I don't think there is any cause for naming names—we all know which OSs I am talking about (and they were not *all* from Redmond, either.

2. This is one nit pick I have with Cocoa's naming scheme for traditional C structures. You can't know intuitively that `NSWindow` is a class, but `NSRect` is not. It is easy enough to teach yourself the difference, but creates undue difficulty for beginners.
3. Ignoring the compiler warnings.
4. See Section 1.6 at the end of this chapter for information on where you can get a copy of **class-dump**
5. `self` in this context seems a little strange, but it makes sense if you think about it. To the class method, the receiver *should* be `self`.
6. The double pointer is C shorthand for a pointer to an array of objects, like `char **argv` versus `char *argv[]`
7. Technically, you could implement your own root class that is not derived from `NSObject`; however, making any objects that were derived from this root class work with the rest of the Foundation or AppKit frameworks would be time consuming. In addition to `performSelector:`, there are many methods implemented by `NSObject` that are used by the Objective-C runtime and standard libraries that would require duplication.
8. Not to dismiss GNUStep and other Unix operating systems