64-Bit Transition Guide for Cocoa

Cocoa > Design Guidelines

2009-03-03

Contents

Tables

Introduction to 64-Bit Transition Guide For Cocoa

Mac OS X is undergoing a gradual transition to a 64-bit model. An 64-bit executable is a process that can support a 64-bit address space. In a 64-bit world pointers are 64 bits (eight bytes) and some integer types, once 32 bits, are now 64 bits. These fundamental changes required for 64-bit processes impact the programmatic interfaces of many of the frameworks of Mac OS X, including Cocoa, in various ways.

This document explains the rationale for the 64-bit transition and describes the 64-bit related changes to the Cocoa API. It also discusses the steps you must take to convert existing Cocoa applications to a 64-bit model.

Important: This is a preliminary version of this document. Future versions of the document may contain modifications and additional information.

Organization of This Document

This document has the following chapters:

- "Moving to 64-Bit Addressing" (page 9) provides background information about the 64-bit initiative for Mac OS X and offers general advice for developers thinking about moving their Cocoa projects to 64-bit addressing.
- "64-Bit Changes To the Cocoa API" (page 13) describes the changes to data types for integers, floating-point values, enumeration constants, and other types of values in the Cocoa API.
- "Converting an Existing Application to 64-Bit" (page 21) explains how to run the script for converting a Cocoa code base to 64-bit and describes modifications you may have to perform manually after the script is run.

See Also

As a prerequisite to this document, read *64-Bit Transition Guide*, which describes in full detail the reasons for the 64-bit transition, the fundamental changes to Mac OS X to support 64-bit addressing, and the general requirements and procedure for building 64-bit executables.

Moving to 64-Bit Addressing

This chapter describes the overall 64-bit initiative for Mac OS X and offers advice and guidelines for moving your Cocoa projects to 64-bit addressing.

Note: This chapter summarizes some of the background information, requirements, and issues related to 64-bit executables presented in *64-Bit Transition Guide*. Refer to that document for a more complete description.

The 64-Bit Initiative

Since version 10.4 (Tiger), Mac OS X has been moving to a model that supports a 64-bit address space. In this model, called LP64, `long` integers and pointers are both 8 bytes (64 bits) instead of 4 bytes. In addition, the `size_t` integer type is 8 bytes instead of 4 bytes. The alignment of these types for LP64 has also increased to 8 bytes. The sizes of all other primitive integer types (`char`, `int`, `off_t`, and so on) remain as they are in the 32-bit model (ILP32), but the alignment of some—namely `long long` and `pos_t`—has increased to 8 bytes for LP64.

On Intel architectures, 64-bit mode also entails an increase both in the number of registers and in their width, as well as a change in the calling conventions to pass arguments in registers instead of on the stack. As a direct consequence of these changes, 64-bit executables on Intel-based Mac OS X systems may see a boost in performance. (The expansion of registers does not happen on the PowerPC architecture, because it was designed for 64-bit computing from the outset.) Although the kernel (Darwin) remains 32-bit, it supports 64-bit software in user space.

All pointers in a 64-bit process are 64 bits; there is no "mixed mode" in which some pointers are 32 bits and others are 64 bits. Consequently, all supporting binaries needed to run a process, including frameworks, libraries, and plug-ins, must be 64-bit capable if the process is to run in a 64-bit address space. All dependencies require porting to 64-bit.

As part of the 64-bit initiative for Mac OS X v10.5 (Leopard), Apple is porting all system frameworks, libraries, and plug-ins to support 64-bit addressing. They are packaged to support both 32-bit and 64-bit executables. Thus, if you have a 32-bit application and a 64-bit application running at the same time, both framework stacks on which the applications have dependencies are loaded into memory. The GCC compiler, linker, debugger, and other development tools have also been modified to support 64-bit addressing. System daemons are also being modified to support 64-bit processes. The Cocoa frameworks as well as the Objective-C runtime and related development tools are part of the porting effort. (The Cocoa changes are described in "64-Bit Changes To the Cocoa API.")

Several changes have also been made to intermediate integer types in lower layers of the system. For example, the underlying primitive type for `CFIndex` in Core Foundation has been changed to be 64-bit while that for `SInt32` in Carbon Core has been changed to remain 32-bit in a 64-bit world. These changes percolate up into higher layers of the system, affecting frameworks where they expose these types in their APIs.

There are several consequences and implications of a transition to a 64-bit address space:

- A 64-bit executable can concurrently access 16 exabytes of virtual address space.
- System memory requirements will more than double, both because of larger data structures and the need for two framework stacks loaded simultaneously.

An important stake of the 64-bit initiative is to maintain binary compatibility for existing 32-bit applications once 64-bit changes are made to system frameworks, libraries, and plug-ins. Sometimes this means keeping the underlying primitive type the same for 32-bit values.

Should I Move My Project to 64-Bit?

For Mac OS X v10.5 (Leopard), just a small percentage of projects have any need to move to a 64-bit address space. Generally, these projects are for applications that require the increased address space for manipulating large data sets, or that need to have random access to data objects exceeding 4 GB. Some examples of applications that fall into this category include those that perform scientific computing, large-scale 3D rendering and animation, data mining, and specialized image processing.

Note: If you have a project that builds a publicly available framework, library, or plug-in, you should port it to 64-bit for Mac OS X v10.5. This is especially true if your framework, library, or plug-in has an API that needs to cover a range of values that cannot be handled with 32 bits. Similarly, if you own a server or daemon that needs to talk with processes that can be either 32-bit or 64-bit, you need to make sure your IPC mechanism can deal with both 32-bit and 64-bit pointers and data structures.

For releases after Mac OS X v10.5 it is expected that more and more software projects—including most consumer applications—will make the transition to 64-bit. There are several reasons for this expectation:

- **Competing platforms**. Microsoft Windows XP already has a 64-bit version and Windows Vista will also be 64-bit capable.
- **Hardware evolution**. As the downward trend in the price of hardware components (such as memory chips) continues, the typical configuration of computers will grow to allow 64-bit computing to become the norm. Moreover, as noted above, you may see performance improvements for 64-executables running on Intel-based Macintosh computers.
- **User demand**. Along with hardware configurations, user expectations will also grow. On a 64-bit capable Mac OS X system with, say, 32 GB of memory, few users will be happy with an application that can access no more than 4 GB of data.

So although your application may not immediately need to make the transition to 64-bit, in a few years' time it will. You can start now to prepare your projects for this transition, proceeding in stages. First, make your projects 64-bit clean by incorporating the new 64-bit types in your code and using the preprocessor conditionals to maintain source compatibility. The project should be able to compile without errors and run. Next make your projects 64-bit enabled by using APIs that express 64-bit quantities. Finally, make your projects 64-bit capable by ensuring that all code paths are capable of dealing with 64-bit quantities.

You can ease the transition process by adopting Mac OS X-native technologies and data types. For example, if you use Core Data (see *Core Data Programming Guide*) to manage your application's data you can reduce the number of parameters you have to consider in adoption. The attribute types that you specify for Core Data entities are the same on 32- and 64-bit platforms. The file format is platform-independent (it is the same whether the host is Intel- or PowerPC-based, 32- or 64-bit); moreover, using the lazy data initialization available with the SQLite store, Core Data allows you to work with data sets whose size is constrained primarily by the host. Together these features make it easier for you to scale seamlessly from 32- to 64-bit.

You can use existing technologies to make your application "universal" by packaging binaries in your application bundle with 64-bit and 32-bit variants combined with variants for PowerPC and Intel architectures. Thus your application could have four-way multi-architecture binaries, with binaries for PowerPC 32-bit, PowerPC 64-bit, Intel 32-bit, and Intel 64-bit. Generally, you will want an application that ships as 64-bit to ship as 32-bit as well, since you want it to run on 32-bit-only hardware, which will be common for many years.

64-Bit Changes To the Cocoa API

The 64-bit changes to the Cocoa API are largely targeted at those parameter and return types required for 64-bit addressing. However, consistency, ease of porting, and "impedance match" are also goals and thus most 32-bit quantities in the API are growing to 64 bits. The Cocoa 64-bit changes fall into three major categories: integers, floating-point values, and enumeration constants.

If you are a creating a new Cocoa application and you want to make it 64-bit capable from the outset, you need to incorporate the API changes into your code. You also might want to maintain a source base that ensures binary compatibility and source compatibility for both 32-bit and 64-bit architectures.

Note: For the procedure for porting *existing* projects to 64-bit, see "Converting an Existing Application to 64-Bit" (page 21).

Integers

The biggest 64-bit change in the Cocoa API is the introduction of the `NSInteger` and `NSUInteger` data types. These two types now replace most occurrences of `int` and `unsigned int` in the framework headers. The parameter and return types that remain as `int` and `unsigned int` in Cocoa header files are unchanged for one of two reasons:

- They need to be a fixed size,
- They need to correspond with those defined by another API, such as file descriptors, Mach ports, and four-character codes.

On 64-bit architectures, `NSInteger` and `NSUInteger` are defined as `long` and `unsigned long`, respectively. To maintain binary compatibility with 32-bit applications, they are declared in the Foundation header file `NSObjCRuntime.h` using the `__LP64__` macro to distinguish between 32-bit and 64-bit builds, as follows:

```
#if __LP64__
    typedef long NSInteger;
    typedef unsigned long NSUInteger;
#else
    typedef int NSInteger;
    typedef unsigned int NSUInteger;
#endif
```

Note: Although these conditional declarations help to preserve binary compatibility when project code is built for a 32-bit or a 64-bit architecture, it does not prevent some compatibility problems in source code. For example, different `printf`-style type specifiers have to be used depending on whether the underlying type is `int` or `long`. For one approach toward resolving this incompatibility, see "Building 32-Bit Like 64-Bit" (page 15).

Additionally for the 64-bit initiative, many new methods have been added to the Cocoa frameworks with "integer" in their names. These methods are counterparts to other methods in the same class with "int" in their names; these methods need to continue dealing with values that are specifically `int`. The "integer" methods have parameter or return types of `NSInteger` or `NSUInteger` while the "int" methods accept or return values of the native types `int` or `unsigned int`. Table 2-1 lists the new methods.

Table 2-1 "Integer" methods added to the Cocoa frameworks

Class	Methods

`NSCoder`	`encodeInteger: forKey:`
	`decodeIntegerForKey:`

`NSUserDefaults`	`setInteger:forKey:`
	`integerForKey:`
	(Note: available since Mac OS X v10.0)

`NSString`	`integerValue`

`NSCell`	`integerValue`
	`setIntegerValue:`

`NSControl`	`integerValue`
	`setIntegerValue:`

`NSScanner`	`scanInteger:`

`NSNumber`	`integerValue`
	`unsignedIntegerValue`
	`initWithInteger:`
	`initWithUnsignedInteger:`
	`numberWithInteger:`
	`numberWithUnsignedInteger:`

`NSActionCell`	`integerValue`

To set limits for the new `NSInteger` and `NSUInteger` types, `NSObjCRuntime.h` also defines the following constants:

```
#define NSIntegerMax LONG_MAX
#define NSIntegerMin LONG_MIN
#define NSUIntegerMax LONG_MAX
```

Building 32-Bit Like 64-Bit

As shown in the previous section, Cocoa defines `NSInteger` and `NSUInteger` conditionally (using the `__LP64__` macro) so that, as long as the project consistently uses the new data types, the underlying primitive type varies according to whether the target architecture is 32-bit or 64-bit.

The `NS_BUILD_32_LIKE_64` preprocessor macro works in a different manner. It declares `NSInteger` to be `long` (instead of `int`) and `NSUInteger` to be `long unsigned int` even on 32-bit portions of the source base.

```
#if __LP64__ || NS_BUILD_32_LIKE_64
    typedef long NSInteger;
    typedef unsigned long NSUInteger;
#else
    typedef int NSInteger;
    typedef unsigned int NSUInteger;
#endif
```

This makes it possible to do something like the following without getting warnings.

```
NSInteger i;
printf("%ld", i);
```

The `NS_BUILD_32_LIKE_64` macro is useful when binary compatibility is not a concern, such as when building an application.

Floating-Point Values

Floating point quantities in the Core Graphics framework (Quartz), which are `float` on 32-bit architectures, are being expanded to `double` to provide a wider range and accuracy for graphical quantities. Core Graphics declares a new type for floating-point quantities, `CGFloat`, and declares it conditionally for both 32-bit and 64-bit. This change affects Cocoa because of its close dependency on Core Graphics. Where a parameter or return value in the Cocoa frameworks is a graphical quantity, `CGFloat` now replaces `float`.

The `CGFloat` changes made in the Foundation and, especially, the Application Kit are so numerous that its easier to point out the methods with floating-point parameters and return types that *don't* change to `CGFloat`; that is, they remain as `float`. These methods fall into certain categories, described in the captions to Table 2-2, Table 2-3, Table 2-4, Table 2-5, and Table 2-6.

Table 2-2 Type-specific parameters and return values

Class	Methods
`NSActionCell`	`floatValue`

`NSCell`	`floatValue`
	`setFloatValue:`

`NSControl`	`floatValue`
	`setFloatValue:`

Class	Methods
NSString	floatValue
NSNumber	floatValue initWithFloat: numberWithFloat:
NSScroller	setFloatValue: knobProportion:
NSPrinter	floatForKey:inTable::
NSUserDefaults	floatForKey: setFloat:forKey:
NSScanner	scanFloat:
NSKeyedArchiver	encodeFloat:forKey:
NSKeyedUnarchiver	decodeFloatForKey:
NSCoder	encodeFloat:forKey: decodeFloatForKey:
NSByteOrder.h	About a dozen "swap" functions.

Table 2-3 Typesetting factors

Class	Methods
NSATSTypesetter	hyphenationFactor hyphenationFactorForGlyphAtIndex: setHyphenationFactor:
NSLayoutManager	hyphenationFactor setHyphenationFactor:
NSTypesetter	hyphenationFactor setHyphenationFactor:
NSParagraphStyle	hyphenationFactor tighteningFactorForTruncation

Table 2-4 Media: frame rate, volume, animation percentage

Class	Methods

`NSAnimation`	`frameRate`
	`setFrameRate:`
	`currentValue`
	`animation: valueForProgress:`

`NSMovieView`	`setRate:`
	`rate`
	`setVolume:`
	`volume`

`NSSound`	`setVolume:`
	`volume`

Table 2-5 Compression factor

Class	Methods

`NSBitmapImageRep`	`getCompression: factor:`
	`setCompression: factor:`
	`TIFFRepresentationUsingCompression: factor:`
	`TIFFRepresentationOfImageRepsInArray: usingCompression: factor:`

`NSImage`	`TIFFRepresentationUsingCompression: factor:`

Table 2-6 Tablet-event pressure and rotation

Class	Methods

`NSEvent`	`pressure`
	`rotation`
	`tangentialPressure`
	`mouseEventWithType: location: modifierFlags: timestamp: windowNumber: context:eventNumber: clickCount:pressure:`

Enumeration Constants

A problem with enumeration (`enum`) constants is that their data types are frequently indeterminate. In other words, `enum` constants are not predictably `unsigned int`. With conventionally constructed enumerations, the compiler actually sets the underlying type based on what it finds. The underlying type can be (signed) `int` or even `long`. Take the following example:

```
type enum {
    MyFlagError = -1,
    MyFlagLow = 0,
    MyFlagMiddle = 1,
    MyFlagHigh = 2
} MyFlagType;
```

The compiler looks at this declaration and, finding a negative value assigned to one of the member constants, declares the underlying type of the enumeration `int`. If the range of values for the members does not fit into an `int` or `unsigned int`, then the base type silently becomes 64-bit (`long`). The base type of quantities defined as enumerations can thus change silently size to accord with the values in the enumeration. This can happen whether you're compiling 32-bit or 64-bit. Needless to say, this situation presents obstacles for binary compatibility.

As a remedy for this problem, Apple has decided to be more explicit about the enumeration type in the Cocoa API. Instead of declaring arguments in terms of the enumeration, the header files now separately declare a type for the enumeration whose size can be specified. The members of the enumeration and their values are declared and assigned as before. For example, instead of this:

```
typedef enum {
    NSNullCellType = 0,
    NSTextCellType = 1,
    NSImageCellType = 2
} NSCellType;
```

there is now this:

```
enum {
    NSNullCellType = 0,
    NSTextCellType = 1,
    NSImageCellType = 2
};
typedef NSUInteger NSCellType;
```

The enumeration type is defined in terms of `NSInteger` or `NSUInteger` to make the base enumeration type 64-bit capable on 64-bit architectures. For Mac OS X v10.5 all enumerations declared in the Cocoa frameworks now take this form. In some cases, an enumeration type is now declared where one had not existed before.

Unfortunately, this change affects type checking of enumeration constants; you can pass any integer value in a method parameter typed as, say, `NSCellType`, not just one of the values in the enumeration. However, the change does allow more specific typing of bit masks, which were previously declared as `unsigned int`. You can now use the `typedef`s for parameters which are bit masks. For instance,

```
- (NSComparisonResult)compare:(NSString *)string options:(unsigned)mask;
```

can now be

```
- (NSComparisonResult)compare:(NSString *)string
options:(NSStringCompareOptions)mask;
```

Other Related Changes (and Non-Changes)

The use of some of the other primitive C types in the Cocoa API are changing in the 64-bit initiative, while others are unaffected. The following summarizes the changes and non-changes:

- `char` and `short` — No changes in the Cocoa API.
- `long` — In versions of Mac OS X prior to version 10.5, the scripting and Apple event part of the Cocoa API used `long` for four-byte codes; these return types and parameter types are now changing to `FourCharCode`.
- `long long` — Prior to Mac OS X v10.5, the `NSFileHandle` class used `long long` for file offsets, which gives a 64-bit value on both 32-bit and 64-bit architectures. This is unchanged.

The Cocoa OpenGL API (including the classes `NSOpenGLContext`, `NSOpenGLPixelBuffer`, `NSOpenGLPixelFormat`, and `NSOpenGLView`) follow the data-type changes made for the C OpenGL framework for Leopard by adopting the standard OpenGL types such as `GLint`, `GLsizei`, and `GLenum` for parameters and return values. These types were chosen in part to maintain binary compatibility under 32-bit (where `long` and `int` are the same size).

The Objective-C runtime API (`/usr/include/objc`) has undergone significant modification for Mac OS X v10.5. Most Cocoa developers don't directly call these functions, so these changes should not affect them. However, if you do use the Objective-C runtime API, you should be aware that there are implications for 64-bit binaries. If you are building your project 64-bit, you cannot use the old Objective-C runtime API.

Converting an Existing Application to 64-Bit

To assist you in converting an existing application project to 64-bit, Apple provides a script that does most of the work. The script performs in-place substitutions on source-code files according to a set of rules. However, after you run the script you still need to look at everything that changed or was flagged and fix it if necessary. This procedure described below works as well for existing framework projects, plug-in projects, or other projects.

The Conversion Procedure

Start by eliminating the build warnings that your project currently generates; this will make it easier to focus on warnings that are specific to 64-bit conversion problems. Also save a copy of the project in case something goes wrong in the conversion process and you need to start over. Then in a Terminal shell run the conversion script `ConvertCocoa64` located in `/Developer/Extras/64BitConversion/`. You can run the script on specific source and header files, as in this example:

```
/Developer/Extras/64BitConversion/ConvertCocoa64 Controller.h Controller.m
CustomView.h CustomView.h
```

Or you can run it on all `.h` and `.m` files in your project, as in this example:

```
/Developer/Extras/64BitConversion/ConvertCocoa64 `find . -name '*.[hm]' | xargs`
```

As the script runs, it does the following things:

- It converts most instances of `int` and `unsigned int` to `NSInteger` and `NSUInteger`, respectively. It doesn't convert `int`s in bit-field declarations and other inappropriate cases. During processing, the script refers to a hardcoded list of exceptions.

Important: The script converts instances of `int` and `unsigned int` that are properly the types of arguments and return values in POSIX calls. You must be aware of these instances and manually revert them (or leave them alone, if that is more appropriate).

- It converts most instances of `float` to `CGFloat`, again leaving untouched those exceptions specified in a hardcoded list.
- It does *not* convert `enum` declarations to types with guaranteed fixed sizes (as described in "Enumeration Constants" (page 17)). You need to fix these declarations by hand.
- It flags (with inline warnings) cases that need to be fixed manually. See "Things to Look For During Conversion" (page 22) for details.

After the script completes processing all files, run the FileMerge development application on each changed file and an unchanged version of the same file. FileMerge visually highlights each modified section of code. After reading "Things to Look For During Conversion" (page 22), evaluate each of these sections plus the warnings generated by the script and make further changes if necessary.

Remember that any 64-bit change you make should help preserve binary compatibility of your project executable on 32-bit systems. You should use the constructs in Table 3-1 to mark off sections of code conditionally for each architecture: PowerPC versus Intel and 32-bit versus 64-bit.

Table 3-1 Macros for architecture conditional code

Macro	Comment
`#if __LP64__`	PowerPC or Intel 64-bit. You should use this for most 64-bit changes.
`#ifdef __ppc__`	PowerPC 32-bit
`#ifdef __ppc64__`	PowerPC 64-bit
`#ifdef __i386__`	Intel 32-bit
`#ifdef __x86_64__`	Intel 64-bit
`#ifdef __BIG_ENDIAN__`	Big endian architectures

Things to Look For During Conversion

When the conversion script processes a source or header file, it often inserts warnings in the code to alert you to changes that you might have to make or pitfalls that you should try to avoid. These warnings are explained in the following sections.

Type Specifiers

Script action: Warns about potential problems; may generate false negatives.

Typically, in 32-bit code you use the `%d` specifier to format `int` values in functions such as `printf`, `NSAssert`, and `NSLog`, and in methods such as `stringWithFormat:`. But with `NSInteger`, which on 64-bit architectures is the same size as `long`, you need to use the `%ld` specifier. Unless you are building 32-bit like 64-bit, these specifiers generates compiler warnings in 32-bit mode. To avoid this problem, you can cast the values to `long` or `unsigned long`, as appropriate. For example:

```
NSInteger i = 34;
printf("%ld\n", (long)i);
```

Table 3-2 lists specifiers for formatting 64-bit types.

Table 3-2 Type specifiers for 64-bit types

Type	Specifier	Comments
`NSInteger`	`%ld` or `%lx`	Cast the value to `long`.
`NSUInteger`	`%lu` or `%lx`	Cast the value to `unsigned long`.

Type	Specifier	Comments
`CGFloat`	`%f` or `%g`	`%f` works for floats and doubles when formatting (but not scanning).
`CFIndex`	`%ld` or `%lx`	Cast the value to `long`.
`long long`	`%lld` or `%llx`	`long long` is 64-bit on both 32-bit and 64-bit architectures.
`unsigned long long`	`%llu` or `%llx`	`unsigned long long` is 64-bit on both 32-bit and 64-bit architectures.

The script also inserts warnings in the related case of scanning functions such as `scanf`, but these introduce an additional subtlety. You need to distinguish between `float` and `double` types, using `%f` for `float` and `%lf` for `double`. Don't use `CGFloat` parameter in `scanf`-type functions; instead use a `double` parameter and assign the `double` value to a `CGFloat` variable. For example,

```
double tmp;
sscanf(str, "%lf", &tmp);
CGFloat imageWidth = tmp;
```

Important: Unless you cast appropriately, the `%lf` specifier (unlike the `%ld` specifier and `long` values) does not correctly represent `CGFloat` values on both 32-bit and 64-bit architectures.

Unkeyed Archiving

Script action: Warns of potential problems

The old style of Cocoa archiving relies on the sequence of encoded and decoded properties rather than keys as the means for identifying archived values; you must decode properties in the same order of their encoding. Developers are encouraged to move away from this old-style archiving to keyed archiving. However, applications may have to read old-style archives so they cannot abandon this key-less archiving altogether. And, from a 64-bit perspective, you cannot change these archives to accommodate larger values because that would make it impossible for older versions of the application to decode the archive.

When reading old-style archives, remember that integers encoded with the "i" and "l" (small L) type-encoding specifiers are 32-bit even on 64-bit architectures. To specify 64-bit integers you must use the "q" type-encoding specifier. But if you type an instance variable as an `NSInteger` value, the following decoding attempt won't work on 64-bit architectures:

```
// Header file
NSInteger bitsPerSample;  // instance variable

// Implementation file
[coder decodeValuesOfObjCTypes:"i", &bitsPerSample];
```

One way to get around this problem is to use a local variable for an intermediate value, then assign this value to the `NSInteger` instance variable after decoding.

```
// Header file
NSInteger bitsPerSample;
// Implementation file
```

```
int tmp;
[coder decodeValuesOfObjCTypes:"i", &tmp];
bitsPerSample = tmp;
```

You can use the same intermediate-variable technique for encoding `NSInteger` values (as 32-bit) as well as decoding them.

Instance variables typed as `CGFloat` present a similar problem since the "f" type-encoding specifier refers to `float`, not `CGFloat`. the workaround for this is the same as for `NSInteger`: use a local `float` variable to store the intermediate value, which you then assign to the `CGFloat` instance variable.

Cocoa is changing most enumerations to be based on the `NSInteger` type, which you should probably do in your own code too. Nevertheless, for old-style archiving you need to archive `enum` constants based on what the explicit or underlying type is.

Avoid using the `@encode` compiler directive in old-style archiving. The `@encode` directive generates a character string with the user-defined type stripped away. For example, during an LP64 build `NSInteger` resolves to `long`, but during a 32-bit build `NSInteger` resolves to `int`. The `NSInteger` semantics have been lost, and this could lead to bugs.

See "Encoding Verification" (page 26) for information on a script that you can run to verify the consistency of old-style archiving code.

Keyed Archiving

Script action: Converts `encodeInt:forKey:` with `encodeInteger:forKey:` (`NSInteger`) and `encodeFloat:forKey:` with `encodeDouble:forKey:` (`CGFloat`).

You should examine the substitutions made by the script to see if any decoded values might be out of range. With keyed archiving it doesn't really matter if you encode integers with `encodeInt:forKey:`, `encodeInt32:forKey:`, or `encodeInt64:forKey:`. But, if on decoding, the archived value is larger than what 32-bit signed integers can hold, an `NSRangeException` is raised.

For most integral values, use of `encodeInteger:forKey:` and `decodeIntegerForKey:` are recommended. For values whose ranges are larger than what 32-bit signed integers can hold, `encodeInt64:forKey:` and `decodeInt64ForKey:` are the more appropriate choice, even on 32-bit systems.

In the rare case where an application wants to read 64-bit values in 32-bit processes, it can use `decodeInt64ForKey:` and be explicit about the type rather than using `decodeIntegerForKey:` (which reads back 32-bit on 32-bit systems).

Archiving Variables With NSNotFound Values

Script action: None.

`NSNotFound` is set to `LONG_MAX` in `NSObjCRuntime.h`, which has different values on 32-bit and 64-bit. Consequently, an variable archived with an `NSNotFound` value on a 32-bit system would have a different value if unarchived on a 64-bit system. In light of this, remove code that writes out `NSNotFound` to archives. If it is already written out, then you should check the 32-bit value explicitly when you read it in, even in a 64-bit application.

Instance Variables

Script action: Converts instance variables declared as `int` and `unsigned int` to `NSInteger` and `NSUInteger`, respectively. If integers are declared smaller than `int` and `unsigned int` (`char` or `short`), the script leaves them alone. Also changes `float` instance variables to `CGFloat` if they correspond to programmatic interfaces that deal in `CGFloat` values.

You should go over modified instance variables and verify if each should be `NSInteger` or `NSUInteger`. In some cases it might not be necessary to grow the size of instance variables because, even if the programmatic interface is 64-bit capable, the storage doesn't have to be. If that is the case, change the type of the instance variable back to `int` or `unsigned int`.

sizeof Operator and @encode Directive

Script action: Converts instances of `sizeof` and `@encode` where the argument is `int`, `unsigned int`, and `float` (to `NSInteger`, `NSUInteger`, and `CGFloat`, respectively). For other cases, generates warnings (with possible false negatives).

Go over each warning and make sure that no conversion needs to made manually.

Functions That Round Off Floating-Point Values

Script action: Converts calls of functions such as `roundf` and `ceilf` to (non-existent) functions of the form `_CGFloatRound`.

Because these functions take `float` arguments, any calls to these with `CGFloat` arguments need to be updated to use the `float`- or `double`-taking parameter variant, as appropriate to the architecture. You should implement `_CGFloatRound` as a macro or inline function that calls, for example, `round` or `roundf` depending on the architecture.

Note: You could also simply replace calls to `roundf` with calls to `round`, and so on. However, in 32-bit code there is a performance cost associated with promoting from `float` to `double` and back to `float`.

Arrays of Integers

Script action: Converts instances of `int[]` and `unsigned int[]` to `NSInteger[]` and `NSUInteger[]`. respectively.

If you have `static` or `static const` arrays of `int` or `unsigned int`, and the arrays are not exposed in API, you might want to change the type back for the memory savings.

Use of long and unsigned long

Script action: Marks each occurrence with a warning; does not convert.

Often leaving the type untouched is appropriate, but in some cases you may want to change it to an `int` to remain 32-bit.

Pointer Casts

Script action: Generates warnings for pointer casts that involve `int` or `float` values.

Limit Constants

Script action: Generates warnings for occurrences of `INT_MAX`, `INT_MIN`, `UINT_MAX`, `FLT_MAX`, and `FLT_MIN` since they might be involved in comparisons involving `long` or `float` values.

Switch (respectively) to the `NSIntegerMax`, `NSIntegerMin`, `NSUIntegerMax`, or the appropriate `LONG_` or `DBL_` variants.

Encoding Verification

As a further step toward ensuring 64-bit capable code, you can run a script that verifies the consistency of old-style (unkeyed) archiving code. This script, named `CoderFormatVerifier`, is installed in `/Developer/Extras/64BitConversion/`.

The script replaces `NSCoder` method invocations with type-checked function calls. On the command line, pass the script a directory—it should be copy of your source directory. The script recursively modifies all the `.m` and `.M` files in-place and then writes the function prototypes to standard output (`stdout`). You can then copy and paste the function prototypes into your prefix header, compile the code, and then inspect any compiler errors that you get. (Link errors are expected because the script only generates function prototypes.) Run the script without any arguments for usage information.

Optimizing Memory Performance

64-bit applications have the potential to perform faster than 32-bit applications due to improvements in modern 64-bit processors. However, 64-bit environments increase the size of pointers and some scalar data, resulting in a larger memory footprint for your application. A larger memory footprint results in increased pressure on processor caches and virtual memory and can adversely affect performance. When developing a 64-bit application, it is critical to profile and optimize your application's memory usage.

For a comprehensive discussion on optimizing memory usage, see the *Memory Usage Performance Guidelines*.

Profile Your Application

Before attempting to optimize your application's memory usage, you should first create standard tests that you can run against both the 32-bit and 64-bit versions of your application. Standardized tests allow you to measure the penalty for compiling a 64-bit version of your application when compared to the 32-bit version. It also provides a way to measure improvements as you optimize your application's memory usage. At least one test should use a minimal footprint (e.g. the application has just been opened and shows an empty document). Other tests should include a variety of different data sizes, including at least one test with a very large data set. A complex application may require tests that cover subsets of its features. The goal for these additional tests is to measure whether the memory usage changes significantly as the type or amount of data changes. If a particular kind of data causes the 64-bit version of your application to use dramatically more memory than its 32-bit counterpart, that is a great place to start looking for improvements.

While both the stack and heap usage increases on 64-bit applications, we recommend you focus your efforts on reducing your application's heap usage; heap usage will typically be much greater than your stack usage. The `heap` tool can be used to discover how much memory your application has allocated on the heap. The `heap` tool will also tell you how many objects of each class that your application has allocated, and the total usage for each class. Focus your efforts on the classes that use the most memory. As in most performance tuning, often a small number of optimizations will result in significant improvements.

Malloc

It is crucial to understand the behavior of `malloc` when you are developing a 64-bit version of your application. In addition to calling `malloc` directly, all objective-C objects are allocated using `malloc`.

Small allocations (less than 512 bytes) are rounded to the next largest multiple of 16 bytes. For example, assume your used the following struct:

```
struct node
{
    node        *previous;
    node        *next;
    uint32_t    value;
};
```

When this structure is compiled for a 32-bit environment, it uses 12 bytes of storage; `malloc` actually allocates 16 bytes. But in a 64-bit environment, this structure takes up 20 bytes, and `malloc` allocates 32! An application that allocates many such nodes would waste a significant amount of memory.

Larger allocations are even more critical. If `malloc` is called to allocate a block larger than 512 bytes, it will round to the next highest multiple of 512 and allocate that much memory. Be particularly cautious with classes or structures that are above 256 bytes of memory in a 32-bit environment. If the process of converting a structure to 64-bit results in something that is just over 512 bytes in size, your application won't be using twice as much memory, but almost *four* times as much — most of it wasted.

Use Appropriate Data Sizes

The `ConvertCocoa64` script described in "Converting an Existing Application to 64-Bit" (page 21) converts most instances of `int` and `unsigned int` to `NSInteger` and `NSUInteger`. `NSInteger` is a 64-bit integer on 64-bit applications, doubling the storage required, but dramatically increasing the range of values.

Type	Range
int	-2,147,483,648 to 2,147,483,647
NSInteger	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807

In most cases, your application will not need this larger range of values. Our recommendation is to choose a C99 representation that accurately reflects the range of values your application requires.

Type	Range
int8_t	-128 to 127
int16_t	-32,768 to 32,767
int32_t	-2,147,483,648 to 2,147,483,647
int64_t	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
uint8_t	0 to 255
uint16_t	0 to 65,535
uint32_t	0 to 4,294,967,295
uint64_t	0 to 18,446,744,073,709,551,615

Choose a Compact Data Representation

Look for situations where you can choose a stronger data representation. For example, assume we stored a calendar date using the following data structure:

```
struct date
{
    int second;
    int minute;
    int hour;
    int day;
    int month;
    int year;
};
```

This structure is 24 bytes long, and when converted to use `NSInteger`, it would take 48 bytes, just for a date! A more compact representation would be to simply store the number of seconds, and convert these as necessary.

```
struct date
{
    uint32_t seconds;
};
```

Pack Data Structures

For performance reasons, compilers typically pad fields of a structure so that they can be quickly accessed. For example, take a look at the following struct:

```
struct bad
{
    char        a;         // offset 0
    int32_t     b;         // offset 4
    char        c;         // offset 8
    int64_t     d;         // offset 16
};
```

While the structure only uses 14 bytes of data, because of padding, it takes up 24 bytes of space. If this structure were allocated using `malloc`, it would take 32 bytes; more space wasted than used for the data.

A better design would be to sort the fields from largest to smallest.

```
struct good
{
    int64_t     d;         // offset 0
    int32_t     b;         // offset 8
    char        a;         // offset 12;
    char        c;         // offset 13;
};
```

Now the structure doesn’t waste any space.

Use Fewer Pointers

Avoid overusing pointers in code. Let's look at a previous example again.

```
struct node
{
    node        *previous;
    node        *next;
    uint32_t    value;
};
```

Only one-third of the memory used is payload; the rest is used for linking. If we compile that same structure into a 64-bit application, the links alone are 80% of the total memory used.

For complex data types that fit within a known data size, you may want to replace pointers with an index instead. For example, if we knew that there would never be more than 65535 nodes in the linked list, we could instead use the following definition instead.

```
struct indexed_node
{
    uint32_t    value;
    uint16_t    next;
    uint16_t    previous;
};
node *nodeList; // pointer to an array of nodes;
```

Using a linked list built around indices, we use significantly less space for links compared to our data. More importantly this example only uses a single pointer. When converted from 32-bit to 64-bit, this example only uses four additional bytes, regardless of the size of the list.

Cache Only When You Need To

Caching previously calculated results is a common way to improve an application's performance. However, it is worth investigating whether caching is really helping your application. As the previous examples have shown, memory usage is much higher on 64-bit systems. If your application relies too much on caching, the pressure it puts on the virtual memory system may actually result in worse performance.

Typical examples of behaviors to avoid include:

- Caching any data that a class can cheaply recompute on the fly.
- Caching data or objects that you can easily obtain from another object.
- Caching system objects that are inexpensive to recreate.

Always test to ensure that caching improves the performance of your application. We recommend building hooks into your application that allow you to selectively disable caching. You can test whether disabling a particular cache has a significant effect on the memory usage or the performance of your application. This is a good place to test different memory usage scenarios.

Use Foundation Objects Wisely

Many Cocoa classes offer a flexible feature set, but to do that, they use more memory than a simpler data structure may provide. For example, if you are using an `NSDictionary` object to hold a single key-value pair, this is significantly more expensive than simply allocating a variable to hold that information. Creating thousands of such dictionaries wastes memory.

Document Revision History

This table describes the changes to *64-Bit Transition Guide for Cocoa*.

Date	Notes
2009-03-03	Updated to add a section on tuning memory usage in 64-bit Cocoa applications.
2009-01-06	Updated to refer to ConvertCocoa64 conversion script.
2007-03-22	New document that explains how to port Cocoa projects to 64-bit addressing.