Cocoa Touch
64ビット移行ガイド

Developer

目次

表、リスト

64ビットCocoa Touchアプリケーションについて

デスクトップオペレーティングシステムのアドレスづけ方式を32ビットから64ビットに移行する際には、OSそのものだけでなく、アプリケーションの64ビット化も重要でした。現在ではiOSも、デスクトップOSと同種のアーキテクチャを備えています。iOS 7およびA7プロセッサの登場により、64ビットプロセッサの特徴を活かしたiOSアプリケーションを構築できるようになりました。これはほとんどあらゆる状況で、同じデバイス上で動作する32ビットアプリケーションよりも性能が向上します。

はじめに

Apple A7プロセッサはまったく異なる2種類の命令セットを備えています。ひとつは32ビットARM命令セットで、Appleの前世代プロセッサに相当します。もうひとつは最新の64ビットARMアーキテクチャにもとづく命令セットです。64ビットアーキテクチャは広大なアドレス空間を管理できますが、これが唯一の（あるいは最も重要な）改善点というわけではありません。最新かつ合理的な命令セットであり、演算レジスタ（整数用、浮動小数点数用）も倍増しています。AppleのLLVMコンパイラは、この新しいアーキテクチャを充分に活用できるよう最適化されています。その結果、64ビットアプリケーションは、一度により多くのデータを高速に処理できるのです。64ビット整数演算や独自のNEON演算を多用するアプリケーションは、特に著しい性能向上を示します。したがって、A7プロセッサ上では、32ビットアプリケーションのままでも従来より高速に動作しますが、64ビットアプリケーションに変換すればさらなる高速化が可能です。

64ビットアプリケーションがiOS上で動作するとき、ポインタは64ビット幅になります。従来は32ビット幅だった整数型も64ビット幅に変わります。システムフレームワーク（特にUIKitやFoundation）で使われる多くのデータ型も同様です。これは見方を変えると、メモリ消費量も増えるということであり、したがって適切に管理しないと、逆に処理性能を損なう恐れがあります。

64ビットデバイス上で稼働するiOSには、32ビット版と64ビット版のシステムフレームワークが併存します。動作するアプリケーションがすべて、64ビットランタイム用にコンパイルされていれば、32ビット版フレームワークはロードされません。したがってメモリ消費量が減り、アプリケーションを迅速に起動できることになります。組み込みアプリケーションはすべて64ビット対応済みなので、それ以外（特にバックグラウンド処理が可能なアプリケーション）もすべて64ビット用にコンパイルすれば、あらゆる点で有利になるでしょう。性能的に問題がないアプリケーションであっても、メモリ効率が向上する結果、さまざまな利点が得られます。

以下、アプリケーションを64ビット対応にする手順を示します。

アプリケーションをiOS 7用に更新し、64ビットバイナリに変換する

Xcode 5.0.1では、32ビット/64ビットのどちらにも対応したアプリケーションを構築できます。ただし、この混成バイナリが動作するのはiOS 5.1.1以降です。64ビットバイナリは、iOS 7.0.3以降が稼働する64ビットデバイス上でのみ実行可能です。既存のアプリケーションは、まずiOS 7で動作するよう更新し、それから64ビットプロセッサ向けに変換する必要があります。第一段階でiOS 7用に更新することにより、非推奨になったコードパスが除去され、新しい方式にもとづくものになります。新規に開発するのであれば、iOS 7をターゲットとして設定し、32ビット版、64ビット版をそれぞれコンパイルしてください。

iOS上で動作する64ビットアプリケーションのアーキテクチャは、OS Xアプリケーションの場合とほぼ同じなので、どちらでも動作する共通のコードベースを容易に作成できます。Cocoa Touchアプリケーションを64ビット版に変換する手順は、OS X上で動作するCocoaアプリケーションの場合と同様です。ポインタや、Cの一般的なデータ型のいくつかが、32ビット幅から64ビット幅に変わります。`NSInteger`型および`CGFloat`型の値を扱うコードは、入念に調べる必要があります。

64ビットランタイム用にアプリケーションをビルドし、コンパイル時に現れる警告を解消するとともに、64ビットであるがゆえの問題がないか確認してください。たとえば次のような件の対処が必要です。

- 関数呼び出しの箇所すべてに、適切なプロトタイプ宣言を用意すること。
- 64ビット値を誤って32ビット型変数に代入することにより、値の切り捨てが起こらないようにすること。
- 64ビット版でも計算が正しく実行されるようにすること。
- 構造体のデータ配置が32ビット版でも64ビット版でも同じになるようにすること（データファイルをiCloudに保存する場合など）。

> **関連する章:** “64ビット版の主な変更点”（8 ページ）、“メモリ性能の最適化”（28 ページ）

64ビットデバイス上でアプリケーションの動作をテストする

iOSシミュレータは、32ビット/64ビットの、両方のデバイスをシミュレートできます。64ビットランタイム用に変換した後、まずはシミュレータ上でテストしてください。特に独自のデータ構造体を用いるコードについて、どちらのアーキテクチャでも正常に読み書きできることかどうか確認するためには、非常に有用です。ただし、実際にアプリケーションを配布するまでの間に、実ハードウェア上でもテストしなければなりません。ランタイムによる違いの中には、この方法でなければ顕在化しない事項もいくつかあるからです。

正常に変換できたら、Instrumentsを使って処理性能やメモリ消費量を調べてください。

関連する章: "アプリケーションを64ビットバイナリに変換"（13 ページ）、"メモリ性能の最適化"（28 ページ）

この文書の使い方

64ビットランタイムでの変更点について"64ビット版の主な変更点"（8 ページ）を読んだ後、"アプリケーションを64ビットバイナリに変換"（13 ページ）で、実際にアプリケーションを変換する手順を調べてください。適切なメモリ消費量に抑えるための作業については"メモリ性能の最適化"（28 ページ）で説明しています。

必要事項

このガイドの内容を理解するためには、アプリケーション開発にある程度慣れている必要があるでしょう。詳しくは『*Start Developing iOS Apps Today*』を参照してください。

場合によっては、このガイドの内容を充分に活かすために、ANSI Cのプログラミングに関して詳しく把握している必要があるでしょう。

関連項目

32ビット/64ビットのアプリケーションバイナリインターフェイスについては、『*iOS ABI Function Call Guide*』に詳しい解説があります。

64ビット版の主な変更点

いくつものコードが連携して動作するためには、その挙動がすべて、標準的な規約に沿ったものでなければなりません。ここでいう規約としては、一般的なデータ型の大きさ（バイト長）や形式、あるコードから別のコードを呼び出すための命令群などがあります。コンパイラはこの規約に従って実装されており、互いに連携して動作するバイナリコードを生成できます。この規約を総称して**「アプリケーションバイナリインターフェイス（ABI、application Binary Interface）」**と呼びます。

iOSアプリケーションは、低レベルのABIと、Objective-Cやシステムフレームワークによって規定されるコーディング規約に依拠します。iOS 7以降、64ビットプロセッサを搭載し、32ビットおよび64ビットのランタイム環境をどちらも提供するiOSデバイスが登場しました。アプリケーションの側から見ると、64ビットランタイム環境は、次の2つの点で32ビット環境とは異なっています。

- 64ビットランタイムでは、Cocoa Touchフレームワーク（およびObjective-C言語そのもの）が用いるデータ型の多くについて、バイト長が増し、メモリ上の境界揃え規則が厳格になっています。詳しくは"データ型に関する変更点"（8 ページ）を参照。
- 64ビットランタイムでは、関数呼び出しの箇所に、適切なプロトタイプ宣言が必須です。詳しくは"関数呼び出しに関する変更点"（11 ページ）を参照。

データ型に関する変更点

CやObjective-Cの組み込みデータ型には、バイト長やメモリ上の揃え境界に関する規定がありません。プラットフォームごとに定義することになっています。したがって、規格の制限内で、ハードウェアやオペレーティングシステムの特性に応じて適切に定めることになります。iOS上の64ビットランタイムでは、組み込みデータ型の多くについて、バイト長を変更しています。Cocoa Touchフレームワークで用いる高レベルのデータ型も同様です。この節では、Objective-Cのコードによく現れるデータ型について、どのように変わったかを説明します。

2つの規約：ILP32とLP64

32ビットランタイムはILP32という規約に基づいています。この規約では、整数型、`long`整数型、ポインタがいずれも32ビット長です。一方、64ビットランタイムはLP64という規約に基づいています。整数型は32ビット長ですが、`long`整数型やポインタは64ビット長です。この規約はOS X上で動作するアプリケーションのABIに適合し（同様に、Cocoa Touchの規約はCocoaで使われるデータ型に適合）、したがって、どちらのオペレーティングシステムでも動作するコードを記述しやすくなっています。

表 1-1に、Objective-Cのコードによく現れる整数型をすべて示します。それぞれについて、バイト長およびメモリ上の揃え境界を載せてあります。太字はLP64とILP32で値が異なる箇所を表します。64ビットランタイム用にコンパイルすると、コードの動作が変わりうる箇所に当たります。コンパイラは、64ビットランタイム用にコンパイルする際、`__LP64__`というマクロを定義するようになっています。

表 1-1 OS XおよびiOSにおける、各種整数型のバイト長と揃え境界

整数型	バイト長（ILP32）	揃え境界（ILP32）	バイト長（LP64）	揃え境界（LP64）
`char`	1バイト	1バイト	1バイト	1バイト
`BOOL`、`bool`	1バイト	1バイト	1バイト	1バイト
`short`	2バイト	2バイト	2バイト	2バイト
`int`	4バイト	4バイト	4バイト	4バイト
`long`	4バイト	4バイト	**8バイト**	**8バイト**
`long long`	8バイト	4バイト	8バイト	**8バイト**
ポインタ	4バイト	4バイト	**8バイト**	**8バイト**
`size_t`	4バイト	4バイト	**8バイト**	**8バイト**
`time_t`	4バイト	4バイト	**8バイト**	**8バイト**
`NSInteger`	4バイト	4バイト	**8バイト**	**8バイト**
`CFIndex`	4バイト	4バイト	**8バイト**	**8バイト**
`fpos_t`	8バイト	4バイト	8バイト	**8バイト**
`off_t`	8バイト	4バイト	8バイト	**8バイト**

変わった点をまとめると、次のようになります。

- ポインタは4バイト長から8バイト長に拡大。
- `long`整数は4バイト長から8バイト長に拡大。`size_t`、`CFIndex`、`NSInteger`も4バイト長から8バイト長に拡大。
- `long long`整数およびこれを基底とするデータ型（`fpos_t`、`off_t`）の揃え境界は4バイトから8バイトに拡大。それ以外の型はすべて、自然な揃え境界。すなわち、構造体の各要素は、その基底データ型のバイト長に対応する間隔に揃えられる。

表1-2に、iOSおよびOS Xでよく使われる浮動小数点数型を示します。組み込みデータ型のバイト長は変わっていませんが、64ビットランタイムでは`CGFloat`型の実体が`float`から`double`に変わり、Quartzその他、Core Graphicsの型を用いるフレームワークにおいて、表現できる値の範囲や精度が改善されています（浮動小数点数型はいずれも自然な揃え境界）。

表 1-2 OS XおよびiOSにおける、浮動小数点数型のバイト長と揃え境界

浮動小数点数型	バイト長（ILP32）	バイト長（LP64）
`float`	4バイト	4バイト
`double`	8バイト	8バイト
`CGFloat`	4バイト	**8バイト**

64ビットARM環境はリトルエンディアン方式になっています。これは、ARMv7アーキテクチャに対応するARMプロセッサを搭載したデバイスで動作する、32ビットiOSランタイムに適合します。

アプリケーションに対するデータ型変更の影響

32ビット/64ビットの両方のランタイム環境で動作するアプリケーションを開発する際には、挙動の違いにより、処理性能の違いや互換性の問題が生じうることを頭に入れておいてください。考慮するべき事項をいくつか示します。

- メモリ消費量の増大

  64ビット版アプリケーションは、32ビット版に比べて多くの基底型のバイト長が増しているため、メモリ消費量が増えます。たとえば連結リストのような単純なデータ構造であっても、64ビット版ではメモリに対する負荷が高くなるのです。より時間をかけて、性能の最適化を施すようにしてください。場合によっては、Objective-Cのクラスに定義するインスタンス変数など、データ構造の設計方針の見直しを要するかも知れません。具体例をいくつか“メモリ性能の最適化”（28 ページ）に示します。

- 32ビット版と64ビット版の両方にまたがるデータのやり取り

  32ビット版と64ビット版の両方を公開すれば、版をまたがってデータをやり取りする必要が生じるかも知れません。たとえばiCloud上に保存したファイルに、32ビットデバイス、64ビットデバイスの両方からアクセスする、というような状況です。あるいは、ネットワークを介してデバイス間でデータをやり取りするゲームも考えられます。どちらのランタイムでも、読み書きするデータはメモリ上の配置が同じ、すなわち、各要素のバイト長とオフセットが一致しなければなりません。

- 計算結果の違い

64ビット整数は、32ビット整数に比べて広い範囲の数値を表現できます。32ビット幅から64ビット幅に変わった整数型を使っているアプリケーションの場合、計算結果が変わってしまうかも知れません。具体的に言うと、演算結果が32ビット整数の最大値を超える場合に、オーバーフローするかしないか、という違いが生じるのです。演算の限界に関する違いはほかにもあります。

- 表現できる数値範囲が小さいデータ型にコピーする際の値切り捨て

  2つのデータ型どうしを安全な方法で変換する代わりに、実体は同じと想定して単純に代入していることもありえます。しかし64ビットランタイムでは、この想定が成り立たないかも知れません。たとえば、`NSInteger`型の値を`int`型の変数に代入する処理は、32ビット版では正常に動作します。どちらも32ビット整数だからです。しかし64ビットランタイムでは異なる型なので、代入によりデータが損なわれる恐れがあります。

関数呼び出しに関する変更点

コンパイラは関数呼び出しの箇所で、呼び出し元から呼び出し先に引数を渡すためのコードを生成します。このコードはABIによって異なり、引数をレジスタに置く方式、メモリ上に確保したスタックに積む方式などがあります。アセンブリ言語でコードを記述するのでなければ、通常、呼び出し規約が問題になることはありません。しかし64ビットランタイムでは、関数がどのように呼び出されるのか理解しておくべき場合があります。引数の数が可変の関数（可変引数関数）は、固定の場合と呼び出し規約が異なるのです。

アプリケーションに対する関数呼び出し規約の違いの影響

64ビットランタイムをターゲットにする場合、関数の呼び出し箇所から、当該関数の正確な定義を参照できるようにする必要があります。したがって、次の点に注意してください。

- すべての関数について、プロトタイプを記述してください。
- 関数をキャストする場合、元の関数と同じシグニチャになるよう配慮しなければなりません。特に、引数の渡し方が異なるキャストは避けてください（可変引数の関数ポインタを、固定引数の関数にキャストするなど）。Objective-Cランタイムのメッセージ送信関数を直接呼び出す箇所には、別の規則を適用します（詳しくは“Objective-Cのメッセージはメソッド関数のプロトタイプを使ってディスパッチする” （24 ページ）を参照）。

Objective-Cに関する変更点

Objective-Cのランタイムを直接操作する低レベルのコードを記述する際、オブジェクトの`isa`ポインタに直接アクセスすることはできなくなりました。代わりにこの情報にアクセスするランタイム関数を使ってください。

64ビットランタイムに関する他の変更点

64ビットARM命令セットは、32ビットの命令セットと大きく異なります。アセンブリ言語で記述したコードは、新しい命令セットを用いて実装し直さなければなりません。さらに、iOSの64ビット関数呼び出し規約も、ARM標準と完全に同じではないので、詳しく把握しておく必要があります。詳しくは『*iOS ABI Function Call Guide*』に詳しい解説があります。

まとめ

64ビット環境でも正常に動作するコードを記述するために必要な、主な事項を整理すると次のようになります。

- 64ビット`long`整数を32ビット整数に代入しない。
- 64ビットポインタを32ビット整数に代入しない。
- 算術演算の際に起こるポインタや`long`整数の切り捨て（その他、整数型の違いによる算術演算上の問題）を避ける。
- データ型のバイト長の違いによって起こる、揃え境界の問題を解消する。
- 32ビットランタイム、64ビットランタイムで共通に用いる構造体は、メモリ上の配置が同じであるようにする。
- アセンブリ言語で記述したコードは、64ビット版の新しい命令コードやランタイムに合わせて書き直す。
- 可変引数の関数を固定引数の関数に（あるいはその逆に）キャストすることは避ける。

アプリケーションを64ビットバイナリに変換

32ビット、64ビットの、両方のランタイム環境に対応したアプリケーションを構築する、大まかな手順を以下に示します。

1. **Xcode 5.0.1をインストールする。**
2. **プロジェクトを開く。**プロジェクトを最新の形に更新するよう促すメッセージが現れます。更新すると、64ビット版で問題になりうる事項を、コンパイル時に警告やエラーとして検出するようになります。
3. **iOS 5.1.1以降をターゲットにするよう、プロジェクト設定を更新する。**iOS 5.1以前であれば、64ビットプロジェクトはビルドできません。
4. **プロジェクトのビルド設定「Architectures」を、「Standard Architectures（including 64-bit）」に変更する。**
5. **64ビットランタイム環境に対応するようアプリケーションを更新する。**この工程では、コンパイラが新たに検出するようになった警告やエラーメッセージが役立つでしょう。もっとも、コンパイラ任せですべて足りるわけではありません。独自に記述したコードについては、以下に説明する事項を参考に調べてください。
6. **実際に64ビットハードウェア上でテストする。**開発にはiOSシミュレータが有用ですが、関数呼び出し規約など、実デバイス上でなければ影響が顕在化しない事項もあります。
7. **Instrumentsを使ってメモリ利用効率を改善する。**
8. **両方のアーキテクチャに対応したアプリケーションを送信して承認を求める。**

以下、Cocoa Touchアプリケーションを64ビットランタイム環境に移植する際、よく発生する問題点について説明します。コードを調べる際の参考にしてください。

ポインタを整数型にキャストしない

ポインタを整数型にキャストしなければならない状況はほとんどありません。ポインタ型を一貫して用いることにより、該当する変数がすべて、アドレスの保持に充分な長さであることを保証できます。

たとえばリスト2-1に示すコード例では、ポインタをint型にキャストし、アドレスに対して算術演算を施しています。このコードは、32ビットランタイムでは正常に動作します。int型とポインタはバイト長が同じだからです。しかし64ビットランタイムの場合、ポインタはint型よりもバイト長が大きく、代入するとデータの一部が欠落してしまいます。やりたいことはポインタをintのバイト数分だけ進めることなので、ポインタ自身をインクリメントする（1を加算する）だけで同じ処理が可能です。

リスト 2-1　ポインタをintにキャストするコード例int

```
int *c = something passed in as an argument....
int *d = (int *)((int)c + 4); // Incorrect.
int *d = c + 1;               // Correct!
```

ポインタを整数型に変換しなければならない場合は、切り捨てが起こらないよう、uintptr_t型として代入先の変数を宣言してください。なお、ポインタに整数演算を施した後、再びポインタに戻すと、揃え境界の規則に反することになりえます。その結果、コンパイラが想定外の挙動を示したり、適切な境界にないポインタにアクセスすることにより、プロセッサの動作が異常になったりする恐れがあります。

データ型を一貫して使う

データ型の使い方が一貫していないと、さまざまな問題が発生します。このような問題の多くについてコンパイラは警告を出しますが、実際にどのようなパターンで発生するのか知っておくのも有用でしょう。

関数を呼び出す際には、戻り値を代入する変数の型を、関数の戻り値型と同じにしてください。戻り値型のバイト長が、これを受け取る変数のそれよりも大きい場合、値は切り捨てられます。リスト2-2に、この問題が実際に発生する簡単なパターンを示します。PerformCalculation関数はlong整数を返します。32ビットランタイムでは、int型もlong型も32ビット長なので、コードそのものは正しくなくても、int型の変数には正常に代入できます。一方、64ビットランタイムであれば、代入の結果、上位32ビットが失われてしまいます。戻り値をlong整数に代入するように変更すれば、どちらのランタイムでも正常に動作します。

リスト 2-2　戻り値を変数に代入する際の切り捨て

```
long PerformCalculation(void);

    int  x = PerformCalculation(); // incorrect
```

```
    long y = PerformCalculation(); // correct!
```

同じ問題が、引数として値を渡す際にも起こります。たとえばリスト 2-3のコードは、64ビットランタイムで実行すると、入力引数が切り捨てられます。

リスト 2-3　入力引数の切り捨て

```
int PerformAnotherCalculation(int input);
    long i = LONG_MAX;
    int x = PerformCalculation(i);
```

リスト 2-4のコードも、64ビットランタイムでは、戻り値が切り捨てられます。関数の戻り値型で表現できる数値の範囲を、実際の戻り値が外れているからです。

リスト 2-4　値を返す際の切り捨て

```
int ReturnMax()
{
    return LONG_MAX;
}
```

以上の例はいずれも、`int`型と`long`型が同じと想定しているのが原因です。ANSI Cの規格はこのような想定をしていませんし、実際、64ビットランタイムでは成り立ちません。プロジェクトを最新の形に更新すると、自動的に`-Wshorten-64-to-32`というコンパイラのオプションが有効になり、値が切り捨てられうる多くの状況で、警告が現れることになります。更新していない場合は、明示的にこのオプションを有効にしてください。状況に応じ、`-Wconversion`オプションも指定するとよいでしょう。より多くのメッセージが現れ、潜在的なエラーが見つかりやすくなります。

Cocoa Touchアプリケーションの場合、以下の整数型が現れている箇所について、使い方が適切かどうか確認してください。

- `long`
- `NSInteger`
- `CFIndex`
- `size_t`（組み込み演算`sizeof`の結果）

さらに、どちらのランタイム環境でも`fpos_t`型や`off_t`型は64ビット長なので、`int`型への代入はできません。

列挙型にも大きさが定義される

LLVMコンパイラでは、列挙型に列挙の大きさを定義できます。したがって、何らかの想定に基づいて記述した場合、実際にはそれよりも大きいことがありうるのです。回避策はこの場合もやはり、データ型の「大きさ」について何の想定もしないことです。列挙値は必ず、適切なデータ型の変数に代入するようにしてください。

型変換に関してよくある問題（Cocoa Touch）

Cocoa Touch、特にCore FoundationおよびFoundationには、Cのデータ型を直列化（シリアライズ）し、あるいはこれをObjective-Cのオブジェクト内に取り込む手段があるので、これについても検討が必要です。

64ビットコードではNSInteger型のバイト長が変わる。Cocoa TouchではNSInteger型が多用されます。これは、32ビットランタイムでは32ビット整数、64ビットランタイムでは64ビット整数です。したがって、戻り値型がNSIntegerのフレームワークメソッドから情報を受け取るためには、NSInteger型の変数を使わなければなりません。

NSInteger型のバイト長がint型と同じ、という想定をしないことはもちろんですが、特に次のような状況では注意を要します。

- NSNumberオブジェクトと別の型との変換。
- NSCoderクラスを用いたデータのエンコード/デコード。特に、64ビットデバイスでNSIntegerをエンコードした後、これを32ビットデバイスでデコードする場合、その値が32ビット整数で表現できる範囲外であれば、デコードメソッドが例外を投げます。このような状況では、ビット幅を明示した整数型を使ってください（“ビット幅を明示した整数データ型を使う” （20 ページ）を参照）。
- フレームワークでNSInteger型と定義された定数の取り扱い。特にNSNotFoundという定数には注意してください。64ビットランタイムでは、この値はint型で表現できる範囲外なので、切り捨てが起きてエラーになります。

64ビットコードではCGFloat型のバイト長が変わる。CGFloatは64ビット浮動小数点数型に変わりました。NSInteger型と同様、CGFloat型がfloat型、あるいはdouble型と同じであると想定してはなりません。一貫してCGFloatを使うようにしてください。リスト 2-5 （16 ページ）に、Core Foundationを使ってCFNumber型の値を生成するコード例を示します。このコードは、CGFloat型のバイト長がfloat型と同じ、という誤った想定をしています。

リスト 2-5　一貫してCGFloat型を用いることに関するコード例

```
// Incorrect.
```

```
CGFloat value = 200.0;
CFNumberCreate(kCFAllocatorDefault, kCFNumberFloatType, &value);

// Correct!
CGFloat value = 200.0;
CFNumberCreate(kCFAllocatorDefault, kCFNumberCGFloatType, &value);
```

整数演算に注意する

数値の切り捨て以外にも整数のバイト幅の違いに起因する問題があります。この節では、コードを書き換える際に役立つガイドをいくつか示します。

符号拡張に関する規則（Cおよびこれから派生する言語）

Cおよびこれに類する言語では、バイト幅の大きな変数に代入する際、符号拡張に関する一連の規則に基づいて、最上位ビットを符号ビットとして扱うか否かを判断します。符号拡張規則は次の通りです。

1. 符号なしの値をバイト幅の大きな型に昇格する際には、（符号拡張せず）上位ビットに0を埋める。
2. 符号つきの値をバイト幅の大きな型に昇格する際には、常に符号拡張を施す（結果の型が符号なしであっても同様）。
3. 定数（`0x8L`のように接尾辞で修飾されている場合を除く）は、その値を収容できる最小のバイト長の整数として扱う。数値が16進表記であれば、コンパイラは`int`、`long`、`long long`のどの型としても扱いうる。さらに`signed`、`unsigned`のどちらにもなりうる。10進表記であれば常に`signed`型として扱う。
4. 同じバイト長の、符号つきの値と符号なしの値の和は、符号なしの値になる。

リスト 2-6に、以上の規則によって思わぬ結果が生じている例を、説明とともに示します。

リスト 2-6　符号拡張の例1

```
int a=-2;
unsigned int b=1;
long c = a + b;
long long d=c; // to get a consistent size for printing.
```

```
printf("%lld\n", d);
```

問題点：実行結果は、32ビットランタイムであれば-1（0xffffffff）となりますが、64ビットランタイムの場合は4294967295（0x00000000ffffffff）となります。これはおそらく、想定していた結果ではないでしょう。

原因：なぜこのような結果になったのでしょうか。まず、2つの数値を加算します。符号つきの値と符号なしの値の和なので、結果は符号なしになります**（規則4）**。その後、よりバイト長が大きい型に昇格しますが、このとき符号拡張は起こりません。

解決法：これを解決し、32ビットランタイムと同じ結果が得られるようにするためには、bをlong整数にキャストする必要があります。すると、bを符号拡張なしで64ビット整数に昇格してから加算することになります。符号つき整数を昇格する場合と同じ挙動になるよう強制するのです。その結果、想定どおり-1という結果が得られます。

リスト 2-7に、これに関連する別の例を示します。

リスト 2-7　符号拡張の例2

```
unsigned short a=1;
unsigned long b = (a << 31);
unsigned long long c=b;


printf("%llx\n", c);
```

問題点：期待する結果（32ビット環境で実行した結果）は0x80000000です。しかし64ビット環境で実行した結果は0xffffffff80000000となります。

原因：なぜ符号拡張されるのでしょうか。まず、シフト演算子を施す時点で、変数aはint型に昇格します。short整数の値はすべて符号つきのint型で表現できるので、昇格した結果も符号つきになるのです。

次に、シフト演算が終了した時点で、結果をlong整数として保存します。このとき、(a << 31)で表される32ビット符号つきの値は、符号拡張した上で64ビット値に昇格するのです**（規則2）**。結果が符号なしであっても符号拡張されることに注意してください。

解決法：最初の値をlong整数にキャストしてから、シフト演算を施すようにしてください。short整数は1回だけ64ビット整数に昇格します（64ビットランタイム用にコンパイルした場合）。

ビットやビットマスクの操作

64ビット幅のビット列やマスクを操作する際、うっかり32ビット値として取得するようなことは避けなければなりません。以下の点に注意してください。

どのデータ型に関しても、特定の長さであることを想定しない。`long`整数型の変数に格納された値をビット列と看做してシフトする場合、`LONG_BIT`という定数を使って、`long`整数のビット長を求めてください。変数のビット長を超えてシフト演算を施した場合の結果は、アーキテクチャによって異なります。

必要に応じて反転マスクを使う。`long`整数に対してビットマスクを適用する場合、32ビットランタイムと64ビットランタイムでビット幅が異なるので注意が必要です。マスクを生成する方法は、0拡張するか1拡張するかに応じて2通りあります。

- 64ビットランタイムで、上位32ビットが0のマスク値を生成したい場合は、通常の固定幅マスクで構いません。符号なし整数と同様に、64ビット幅に拡張されるからです。
- 一方、上位32ビットが1のマスク値を生成したい場合は、リスト2-8のように、必要なマスク値を反転し、さらにこれをビット単位で反転する形で記述してください。

リスト 2-8　マスクを反転して符号拡張するコード例

```
function_name(long value)
{
    long mask = ~0x3; // 0xfffffffc or 0xfffffffffffffffc
    return (value & mask);
}
```

64ビット環境では、マスクの上位32ビットに1が充填されることに着目してください。

バイト長や揃え境界が固定のデータ構造体を生成する

32ビット版および64ビット版のアプリケーション間でデータを共有する場合、どちらでも同じ配置のデータ構造体が必要になるかも知れません。そのデータをファイルに保存する、あるいはネットワークを介して、ランタイム環境が異なるデバイスに転送するような場合です。さらに、32ビットデバイスに保存したデータを、64ビットデバイスに取り込んで使う、という状況も考慮しなければなりません。データの相互利用性を確保する必要があるのです。

ビット幅を明示した整数データ型を使う

C99規格には、ハードウェアアーキテクチャによらず、所定のバイト長であることが保証された組み込みデータ型があります。データが固定長でなければならない場合、あるいは変数が取りうる値の範囲が限定されている場合は、このようなデータ型を使ってください。適切なデータ型を選択すれば、固定幅でメモリ中に格納できるだけでなく、必要以上に大きなメモリ領域を確保することも回避できます。

表 2-1に、ビット幅を明示したC99のデータ型と、それぞれが取りうる値の範囲を示します。

表 2-1 ビット幅を明示したC99の整数データ型

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

64ビット整数型の揃え境界に注意する

64ビットランタイムでは、64ビット整数型の揃え境界がすべて、4バイト境界から8バイト境界に変わりました。構造体の各要素がビット幅を明示した整数型であっても、メモリ上の配置がどちらのランタイムでも同じになるとは限りません。 リスト 2-9に、各要素がビット幅を明示した整数型である構造体の例を示します。

リスト 2-9 構造体における64ビット整数の揃え境界

```
struct bar {
    int32_t foo0;
    int32_t foo1;
    int32_t foo2;
    int64_t bar;
```

```
};
```

このコードを32ビットコンパイラでコンパイルした場合、要素barの位置は、構造体の先頭から12バイト目以降になります。同じコードを64ビットコンパイラでコンパイルすると、要素barの位置は、構造体の先頭から16バイト目以降になります。4バイトのパディングをfoo2の後に入れて、barを8バイト境界に揃えるのです。

データ構造体を新規に定義する際には、揃え境界の値が大きい要素から順に並べるようにしてください。こうするとパディングバイトが最小限で済みます。既存の構造体をそのまま使う場合、64ビット整数の揃え境界が変わってしまうのであれば、プラグマを使って適切な揃え境界になるよう強制しなければなりません。リスト2-10に、同じデータ構造体を、各要素が32ビット境界に揃うよう強制した例を示します。

リスト 2-10　プラグマを使って揃え境界を強制した例

```
#pragma pack(4)
struct bar {
    int32_t foo0;
    int32_t foo1;
    int32_t foo2;
    int64_t bar;
};
#pragma options align=reset
```

この方法はどうしても必要な場合にのみ使ってください。揃え境界に合っていない要素にアクセスすると、処理性能が損なわれます。32ビット版で既に使っているデータ構造体と、後方互換性を維持しなければならないような場合に活用するとよいでしょう。

メモリ割り当てにsizeof演算子を使う

バイト長を明示的に指定してmallocを実行する（例：malloc(4)）ことは避けてください。割り当てようとする構造体や変数の正確な長さはsizeof演算子で求めます。コード中、mallocを呼び出している箇所で、すぐ後にsizeofが現れていない箇所を検索し、修正するとよいでしょう。

書式指定文字列をどちらのランタイムでも動作するよう修正する

printfなどの印字関数を、どちらのランタイムでも同じ動作になるよう記述するためには、データ型の違いを吸収するために、技巧を要することがあります。ポインタと同じ長さの整数（uintptr_t）その他の標準型について、この問題を解消するためには、ヘッダファイルinttypes.hに定義されている各種のマクロを使うとよいでしょう。

各種のデータ型に応じた書式文字列を表2-2に示します。ヘッダファイルinttypes.hに定義されている型については表 2-3を参照してください。

表 2-2 標準的な書式文字列

型	書式文字列
int	%d
long	%ld
long long	%lld
size_t	%zu
ptrdiff_t	%td
ポインタ	%p

表 2-3 inttypes.hに追加定義されている書式文字列（Nは数値）

型	書式文字列
int[N]_t（int32_tなど）	PRId[N]（PRId32など）
uint[N]_t	PRIu[N]
int_least[N]_t	PRIdLEAST[N]
uint_least[N]_t	PRIuLEAST[N]
int_fast[N]_t	PRIdFAST[N]
uint_fast[N]_t	PRIuFAST[N]
intptr_t	PRIdPTR
uintptr_t	PRIuPTR

型	書式文字列
intmax_t	PRIdMAX
uintmax_t	PRIuMAX

たとえばintptr_t型変数（ポインタと同じ長さの整数）とポインタの値を印字する場合、リスト2-11のようなコードになります。

リスト 2-11　アーキテクチャに依存する型の印字

```
#include <inttypes.h>
void *foo;
intptr_t k = (intptr_t) foo;
void *ptr = &k;

printf("The value of k is %" PRIdPTR "\n", k);
printf("The value of ptr is %p\n", ptr);
```

関数や関数ポインタの取り扱いに注意する

関数呼び出しの処理方法も、64ビットランタイムと32ビットランタイムでは違いがあります。中でも重要なのは、可変引数のプロトタイプを持つ関数の場合、引数を受け渡しする命令の並びが、固定引数の場合と違っている点でしょう。リスト2-12には、2つの関数のプロトタイプが記述されています。ひとつ目の関数（fixedFunction）は、引数として常に整数を2つ取ります。もうひとつの関数は引数の数が可変（ただし2つ以上）です。32ビットランタイムの場合、リスト 2-12の関数はどちらも、よく似た命令の並びで引数データを取得します。一方、64ビットランタイムの場合は、まったく異なる規約に基づいてコンパイルするようになっています。

リスト 2-12　関数の種類による呼び出し規約の違い（64ビット環境）

```
int fixedFunction(int a, int b);
int variadicFunction(int a, ...);

int main
{
    int value2 = fixedFunction(5,5);
    int value1 = variadicFunction(5,5);
```

```
}
```

64ビットランタイムでは呼び出し規約がより厳密になっているため、呼び出し元が渡した引数を呼び出し先が必ず取得できるよう、正しい方法で呼び出さなければなりません。

関数プロトタイプをすべて宣言する

最新の形に更新したプロジェクト設定でコンパイルする場合、プロトタイプが明示的に宣言されていない関数を呼び出そうとするとエラーになります。関数プロトタイプを宣言して、可変引数かどうか、コンパイラが判断できるようにしてください。

関数ポインタを用いるためには適切なプロトタイプを要する

関数ポインタを受け渡しするコードでは、呼び出し規約が一貫している必要があります。引数並びも常に同じでなければなりません。可変引数の関数を固定引数の関数に（あるいはその逆に）キャストすることは避けてください。リスト2-13に、問題のある関数呼び出しの例を示します。呼び出し規約が異なる関数ポインタにキャストしているため、呼び出し元が引数を渡す方法と、呼び出し先が引数を受け取る方法が適合しません。その結果、アプリケーションがクラッシュしたり、想定外の挙動になったりする恐れがあります。

リスト 2-13　可変引数の関数を固定引数の関数にキャストしてエラーが生じている例

```
int MyFunction(int a, int b, ...);


int (*action)(int, int, int) = (int (*)(int, int, int)) MyFunction;
action(1,2,3); // Error!
```

Important: このように関数をキャストしても、コンパイラはエラーや警告を出しません。また、iOSシミュレータ上では正常に動作しているように見えます。出荷に先立ち、必ず実デバイス上でテストするようにしてください。

Objective-Cのメッセージはメソッド関数のプロトタイプを使ってディスパッチする

キャストに関する上述の規則には例外があります。Objective-Cのランタイムに実装されている、`objc_msgSend`その他のメッセージ送信関数を呼び出す場合です。メッセージ関数のプロトタイプは可変引数の形式ですが、Objective-Cランタイムから呼び出されるメソッド関数のプロトタイプは、こ

れとは異なります。Objective-Cランタイムは、メソッドを実装する関数に直接ディスパッチするので、先に説明したように、呼び出し規約が一致しません。そのため、objc_msgSend関数を、呼び出されるメソッド関数に応じたプロトタイプにキャストする必要があるのです。

リスト2-14に、低レベルのメッセージ関数を使ってメッセージをオブジェクトにディスパッチする、適切な書き方を示します。この例で、doSomething:メソッドは引数が1つであり、可変引数の形式ではありません。このコードは、メソッド関数のプロトタイプを使って、objc_msgSend関数をキャストします。ただし、メソッド関数は常に第1引数、第2引数として、id変数およびセレクタを取ることに注意してください。objc_msgSend関数を関数ポインタにキャストした後、同じ関数ポインタを介してメッセージの呼び出しをディスパッチします。

リスト 2-14 キャストの仕組みを使ってObjective-Cのメッセージ送信関数を呼び出す例

```
- (int) doSomething:(int) x { ... }
- (void) doSomethingElse {
    int (*action)(id, SEL, int) = (int (*)(id, SEL, int)) objc_msgSend;
    action(self, @selector(doSomething:), 0);
}
```

可変引数関数の呼び出しに注意する

可変引数リスト（varargs）には、各引数の型に関する情報がありません。また、引数が自動的に、よりバイト長の大きい型に昇格することもありません。引数として渡されるデータの型を識別する必要がある場合は、書式文字列その他、型情報をvarargs関数に渡す手段を組み込んでください。呼び出し元関数がこの情報を適切に渡せない（あるいはvarargs関数が正しく解釈できない）場合、正しい結果が得られません。

特に、varargs関数がlong整数を想定しているところに32ビット値が渡された場合、varargs関数は、渡された32ビット値と、（実際には次の引数である）32ビット分の「ごみ」から成る、64ビット値が渡されたものとして扱います。同様に、varargs関数がint型の値を想定しているところにlong整数が渡された場合、データの半分だけを引数とし、残りは次の引数の値として扱うことになります（たとえばprintf方式の書式文字列を誤用した場合に発生）。

Objective-Cのポインタに直接アクセスしない

オブジェクトのisaフィールドに直接アクセスするコードは、64ビットランタイム上では正常に動作しません。isaフィールドに、ポインタそのものは保持しないようになりました。代わりに、ポインタに結びついたあるデータを収容し、余ったビットで他のランタイム情報を保持するようになっています。この最適化により、メモリ効率と処理性能が向上しています。

オブジェクトの`isa`フィールドを読み取るためには、`class`プロパティ、または`object_getClass`関数を使います。オブジェクトの`isa`フィールドに値を書き込むには、`object_setClass`関数を使います。

Important: iOSシミュレータではこのエラーが発生しないので、必ず実ハードウェア上でテストするようにしてください。

組み込みの同期プリミティブを使う

アプリケーションに独自の同期プリミティブを実装して、性能改善を試みることもあるでしょう。iOSのランタイム環境には、稼働する各CPU向けに、適切に最適化されたプリミティブ群が揃っています。アーキテクチャに関わる機能が追加される都度、このランタイムライブラリも更新されます。したがって、既存の32ビットライブラリがこのランタイムライブラリを使っていたとすれば、何もしなくても、64ビット環境に加わった新しいCPU機能の恩恵を得ることができます。一方、旧アプリケーションで独自のプリミティブを実装していた場合、組み込みプリミティブよりも数桁劣る命令群やパスを使うことになるかも知れません。したがって、常に組み込みプリミティブを使うようにしてください。

仮想メモリのページ長をハードコードしない

多くの場合、仮想メモリのページ長を意識して処理する必要はありませんが、バッファ割り当てや、ある種のフレームワーク関数を呼び出すために、この情報を要することがあります。iOS 7以降、32ビットランタイムでも64ビットランタイムでも、デバイスによってページ長が違うようになりました。したがって、必ず`getpagesize()`関数でページ長を取得するようにしてください。

コードを位置非依存にする

64ビットランタイム環境では、PIE（位置非依存実行形式、Position-Independent Executables）しか動作しません。最新のアプリケーションは、自動的に位置非依存の形でビルドされます。静的にリンクするライブラリやアセンブリコードなど、位置非依存コードとしてビルドできなくなるような要因がある場合、これを64ビットランタイム用に移植するためには、該当するコードの更新が必要です。32ビットアプリケーションについても、ぜひ位置非依存にするようお勧めします。

詳しくは『*Building a Position Independent Executable*』を参照してください。

32ビット環境でも動作するようにする

現状、64ビットランタイム環境向けのアプリケーションは、32ビット環境でも動作するようにしなければなりません。どちらの環境でも良好に動作するよう作成する必要があるのです。通常は、同じ設計で、どちらの環境でも動作するようにします。しかし場合によっては、それぞれのランタイム環境向けに、別々に設計しなければならないかも知れません。

たとえば、コード全体を通して64ビット整数を用いることが考えられます。どちらの環境でも64ビット整数型は使えますし、常に同じ整数型を使うようにすれば、見通しのよい設計になるでしょう。しかしこうすると、32ビット環境では動作が遅くなります。64ビットプロセッサは、32ビット整数演算とほぼ同じ速度で64ビット整数の演算が可能ですが、32ビットプロセッサの64ビット演算はどうしても性能が出ないのです。さらに、どちらの環境でも、変数に割り当てるメモリ量は増えてしまいます。したがって、想定される範囲の値が保持可能な整数型を使うようにしてください。32ビット整数で演算が可能ならば、32ビット整数を使うということです。詳しくは"ビット幅を明示した整数データ型を使う"（20 ページ）を参照してください。

メモリ性能の最適化

64ビットプロセッサにはさまざまな改善が施されているため、64ビットアプリケーションは、32ビットアプリケーションよりも高速に動作します。その一方で、64ビットランタイムではポインタやいくつかのスカラ型のバイト長が増え、メモリ消費量が増しています。その結果、プロセッサキャッシュや仮想メモリに対する負荷が増し、性能的に不利になる可能性があります。64ビットアプリケーションを開発する際には、メモリ消費量を測定し、最適化を図ることが大切です。

メモリ消費量の最適化に関する包括的な解説が、『*Memory Usage Performance Guidelines*』に載っています。

アプリケーションのプロファイルを取得する

メモリ消費量の最適化を施す前に、32ビット版、64ビット版のどちらに対しても実行できる、標準的なテストルーチンを作成してください。こうしておけば、32ビット版に比べ、64ビット版としてコンパイルしたことによりどの程度の犠牲が生じているか、明らかになります。また、最適化による改善の度合いを評価する基準にもなります。少なくとも1件、メモリ消費量が最小限になるテストを設けてください。たとえば、空の文書を開いて表示するだけのテストです。それ以外に、さまざまな長さのデータを扱うテストを用意してください。非常に大きなデータを処理するテストも1件は必要です（複雑なアプリケーションでは、機能ごとに分割し、それぞれについてテストデータを用意する必要があるかも知れません）。このテストの目的は、データの種類や量の違いにより、メモリ消費量が変化するかどうか把握することです。ある種類のデータを用いたとき、32ビット版に比べて64ビット版のメモリ消費量が著しく増すのであれば、そこは改善の余地が大きいことになります。

メモリ消費に関してよくある問題点

以下、メモリ消費に関してよくある問題とその対処法を示します。

実質的なデータ量が少ないFoundationオブジェクトはコストが高い

Foundationクラスの多くは順応性の高い機能群を備えていますが、その代償として、単純なデータ構造に比べてメモリを多く消費する傾向があります。たとえば`NSDictionary`オブジェクトに保持するキーと値の組が1件だけであれば、単にデータを保持する変数を割り当てる方法に比べ、著しくメモ

リ負荷が高くなります。このような辞書オブジェクトを多数生成することは、メモリの無駄遣いです。Foundationオブジェクトの使い方が適切でない、という問題は、目新しいものではありませんが、64ビットランタイムの場合、その弊害はより大きくなります。

データの配置をコンパクトにする

データの表現方法を工夫できないか検討してください。たとえば日付データを保持する、次のような構造体を考えてみましょう。

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

この構造体は24バイト長でしたが、64ビットランタイムでは、（たかが日付のために）48バイトを占有することになります。一方、ANSI Cの`time_t`型のように、ある時点からの秒数を保持するようにすれば、よりコンパクトになります。必要に応じて日付/時刻の値に変換すればよいのです。

```
struct date
{
    time_t seconds;
};
```

なお、64ビットランタイム用にコンパイルすると、`time_t`型はバイト長が変わります。

データ構造体をパックする

コンパイラは、構造体の各要素の境界を揃えるため、パディングを追加することがあります。たとえば次の構造体を考えてみましょう。

```
struct bad
{
    char        a;          // offset 0
```

```
    int32_t     b;          // offset 4
    char        c;          // offset 8
    int64_t     d;          // offset 16
};
```

この構造体は14バイトのデータからなりますが、パディングが入るため、24バイト分の空間を占めます。

このような場合、揃え境界が大きいものから順に並べ換えるとよいでしょう。

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

こうすればパディングは生じません。

できるだけポインタを使わない

ポインタは使いすぎないようにしてください。たとえば次のデータ構造体を考えてみましょう。

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

これを32ビットランタイム用にコンパイルした場合、12バイトのうち実質的なデータは4バイトだけで、残りはリンク用に使われます。同じ構造体を64ビットランタイム用にコンパイルすると20バイトを占めるようになり、リンク用だけで80%を占めることになります。配列や集約型を使い、ポインタではなくインデックスを保持することも検討してください。

メモリ割り当ての際には境界揃えのためにパディングが生じる

`malloc`関数を直接呼び出す（あるいはObjective-Cのオブジェクトを割り当てるため間接的に呼び出す）と、データの境界揃えのため、要求した以上のメモリが割り当てられることがあります。Cの構造体に用いるメモリは、個別に割り当てるよりも、大きなメモリブロックをいくつかまとめて割り当てる方が効率的かも知れません。

キャッシュ化は本当に必要な場合に限定する

性能向上の手段として、一度計算した値をキャッシュ保存する方法がよく用いられます。しかし、実際に効果があるのか、調査する必要があるかも知れません。先の例からも分かるように、64ビットシステムではメモリ消費量が増大します。キャッシュを多用すると仮想メモリシステムを圧迫し、かえって性能の妨げになっていることがあります。

次のようなデータについては、キャッシュを使わない方がよいでしょう。

- いつでも低負荷で再計算できるデータ
- 他のオブジェクトから容易に取得できるデータやオブジェクト
- 容易に再生成できるシステムオブジェクト
- `mmap()`を介してアクセス可能な読み込み専用データ

キャッシュの効果で実際に性能が向上しているか、必ずテストするようにしてください。フックを組み込んで、選択的にキャッシュを無効にすることも可能です。これを利用して、ある特定のキャッシュを無効にしたとき、メモリ消費量や処理性能に大きな違いがあるかどうかテストできます。キャッシュのアルゴリズムについては、さまざまなデータを用いてテストするようにしてください。

書類の改訂履歴

この表は「*Cocoa Touch 64ビット移行ガイド*」の改訂履歴です。

日付	メモ
2014-02-11	時刻を扱う例を、time_t型を使うように変更しました。
2013-10-22	Xcode 5.0.1に関する情報を追加して改訂しました。
2013-09-18	新規資料。Cocoa Touchプロジェクトを更新して、64ビットランタイムに対応する手順を説明しています。

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
U.S.A.

Apple Japan
〒106-6140 東京都港区六本木 6
丁目10番1号 六本木ヒルズ
http://www.apple.com/jp

Offline copy. Trademarks go here.