改訂2版

Objective-C 逆引きハンドブック

林 晃◆著

目的からObjective-Cの機能が引ける
逆引きリファレンスの決定版!

iOSアプリ開発者の必携書!

iOS 5
Xcode 4.3
対応!

ARC(Automatic Reference Counting)に対応!

24時間無料でサンプルデータをダウンロードできます。

C&R研究所

改訂2版

Objective-C 逆引きハンドブック

林 晃◆著

iOS 5
Xcode 4.3
対応!

C&R研究所

■権利について

- Objective-C、Mac OS、OS X、iPhone、iPad、iPod touch、Xcode、iTunesは、米国および他の国々で登録されたApple Inc.の商標です。
- その他、本書に記述されている社名・製品名などは、一般に各社の商標または登録商標です。
- 本書では™、©、®は割愛しています。

■本書の内容について

- 本書は著者・編集者が実際に操作した結果を慎重に検討し、著述・編集しています。ただし、本書の記述内容に関わる運用結果にまつわるあらゆる損害・障害につきましては、責任を負いませんのであらかじめご了承ください。
- 本書で紹介している操作の画面は、Mac OS X 10.7.3とXcode 4.3.2、iOS SDK 5.1を基本にしています。他の環境では、画面のデザインや操作が異なる場合がございますので、あらかじめご了承ください。

■サンプルについて

- 本書で紹介しているサンプルは、C&R研究所のホームページ(http://www.c-r.com)からダウンロードすることができます。ダウンロード方法については、5ページを参照してください。
- サンプルデータの動作などについては、著者・編集者が慎重に確認しております。ただし、サンプルデータの運用結果にまつわるあらゆる損害・障害につきましては、責任を負いませんのであらかじめご了承ください。
- サンプルデータの著作権は、著者及びC&R研究所が所有します。許可なく配布・販売することは堅く禁止します。

●本書の内容についてのお問い合わせについて

この度はC&R研究所の書籍をお買いあげいただきましてありがとうございます。本書の内容に関するお問い合わせは、FAXまたは郵送で「書名」「該当するページ番号」「返信先」を必ず明記の上、次の宛先までお送りください。お電話や電子メール、または本書の内容とは直接的に関係のない事柄に関するご質問にはお答えできませんので、あらかじめご了承ください。

〒950-3122 新潟県新潟市北区西名目所4083-6　株式会社 C&R研究所　編集部
FAX 025-258-2801
「改訂2版 Objective-C逆引きハンドブック」サポート係

PROLOGUE

本書の初版が出た当時、iOSはまだiPhone OSと呼ばれ、iPhoneとiPod touch専用のOSでした。改訂2版を執筆している現在、iPhone OSはiOSと呼ばれるようになり、iOSで動くiPadも登場しました。そして、当時よりもはるかに多くの場面で使用されるようになり、膨大な数のアプリがApp Storeに登場しました。その数は現在も増え続けていっています。その膨大な数のアプリを開発するための標準的な開発言語である「Objective-C」は、ますます注目を集めています。

本書は「Objective-Cではどう書けばよいのか?」ということから調べることができる逆引き辞典です。各項目では、「具体的にはどう書くのか?」「実行すると、どうなるのか?」ということがわかるように、1つのプログラムとして完結するサンプルコードを用意しました。サンプルコードは、できるだけシンプルにしました。Objective-Cの辞典として使っていただければ幸いです。

本書は、Objective-Cの初心者から上級者まで、すべての人の役に立てるように書きました。他の言語での開発経験がある人ならば、CHAPTER 01とCHAPTER 02を読むだけでも、どのような言語かわかると思います。

本書は、Objective-Cという言語が持つ言語としての機能と「Foundation」という基本フレームワークにフォーカスを当てています。開発においてはOSの持つ機能を引き出すことも重要ですが、そのプログラムが「何をするのか」ということを記述できることが最も大切であり、数学的なアルゴリズムやデータ処理など、ユーザーの目に触れない部分が非常に重要です。そういった部分で効率的にObjective-Cの機能を引き出せるように、生産性の高い開発ができるように、すでにある機能を再開発しないように、ということを意識した内容構成になっています。

CHAPTER 01では、ツールの使い方やTips、すでにあるC言語やC++言語で書かれた資産と組み合わせる方法について書いています。CHAPTER 02とCHAPTER 03では、Objective-C全般の事柄について書いています。CHAPTER 04からCHAPTER 13では、「Foundation」フレームワークの機能をカテゴリ別に解説しています。最後に、APPENDIXとして、実際のアプリ開発にどう応用するのか、どう利用するのかということを紹介しています。

iOSやMac OS X用のアプリを開発するときには、本書で解説している事柄以外に幅広い知識が必要になります。しかし、それらをObjective-Cで記述するときには、必ず本書で解説している事柄に触れると思います。開発の現場で本書が役に立つことができれば幸いです。

Objective-Cは、非常におもしろい言語です。動的言語の特徴を持ち、静的言語の効率性を持つ言語です。どうぞお楽しみください。

そして、最後に、本書を執筆するにあたってスタッフの皆様をはじめ、お世話になった皆様に深く感謝を申し上げます。本書がObjective-Cを使う読者の皆様に少しでもお役に立てば、筆者としてこれ以上の幸せはありません。読者の皆様の開発したアプリの発展を心よりお祈り申し上げます。

2012年4月

アールケー開発 代表　林 晃

本書について

開発環境について

本書では、次のような開発環境を前提にしています。

- OS：Mac OS X 10.7.3
- Xcode：4.3.2
- iOS SDK：5.1

特に記載がない場合の動作環境は、次のような環境を想定しています。特定のバージョンが必要な場合については、その都度、記載しています。

- Mac OS X：Mac OS X 10.6以降（Mac OS X 10.7以降を推奨）
- iOS：iOS 4.0以降（iOS 5.0以降を推奨）

本書のサンプルコードは記載がない場合は、ARC（Automatic Reference Counting）を使用することを前提にしています。ARCの動作環境は、次のようになっています。

- Mac OS X 10.7以降（64ビットバイナリ）
- iOS 5.0以降

また、下記の環境では一部制限事項がありますが、ARCが動作します。

- Mac OS X 10.6.x（64ビットバイナリ）
- iOS 4.x

なお、「@autoreleasepool」文が使用できないコンパイラ（古いコンパイラ）などでは、代わりに「NSAutoreleasePool」クラスを使用するように変更する必要があります。

本書の表記方法

本書の表記についての注意点は、次のようになります。

▶メソッドの種類

メソッドの種類の「+」は「クラスメソッド」を表しています。「-」は「インスタンスメソッド」を表しています。

▶クラスの表記方法について

複数のクラスから利用される頻度が高いいくつかのクラスについては、クラス名ではなく、そのクラスの目的を元にした名称で表記しているものがあります。これには、次ページの表のようなものがあります。

クラス名	表記方法
NSString	文字列
NSMutableString	変更可能な文字列
NSArray	配列
NSMutableArray	変更可能な配列
NSDictionary	辞書
NSMutableDictionary	変更可能な辞書
NSSet	セット
NSMutableSet	変更可能なセット
NSOrderedSet	順序づけされたセット
NSMutableOrderedSet	変更可能な順序付けされたセット
NSIndexSet	インデックスセット
NSMutableIndexSet	変更可能なインデックスセット

▶メソッドの定義

メソッドの定義では、各項目で使用しているメソッドのプロトタイプ宣言を記載しています。メソッドの戻り値の型や各引数の型を確認してください。

▶サンプルコードの中の▼について

本書に記載したサンプルコードは、誌面の都合上、1つのサンプルコードがページをまたがって記載されていることがあります。その場合は▼の記号で、1つのコードであることを表しています。

サンプルファイルのダウンロードについて

本書のサンプルデータは、C&R研究所のホームページからダウンロードすることができます。本書のサンプルを入手するには、次のように操作します。

❶「http://www.c-r.com/」にアクセスします。
❷トップページ左上の「商品検索」欄に「105-4」と入力し、[検索]ボタンをクリックします。
❸検索結果が表示されるので、本書の書名のリンクをクリックします。
❹書籍詳細ページが表示されるので、[サンプルデータダウンロード]ボタンをクリックします。
❺下記の「ユーザー名」と「パスワード」を入力し、ダウンロードページにアクセスします。
❻「サンプルデータ」のリンク先のファイルをダウンロードし、保存します。

サンプルのダウンロードに必要な
ユーザー名とパスワード

ユーザー名 **k2obj**
パスワード **5r8ka**

※ユーザー名・パスワードは、半角英数字で入力してください。また、「J」と「j」や「K」と「k」などの大文字と小文字の違いもありますので、よく確認して入力してください。

サンプルコードの利用方法

サンプルファイルは、CHAPTERごとのフォルダの中に、項目番号のフォルダに分かれています。COLUMNのサンプルについては、「項目番号-COLUMN」（COLUMNのサンプルが複数場合は、末尾に連番）としています。サンプルはZIP形式で圧縮してありますので、解凍してお使いください。

それぞれのフォルダ内には、プロジェクトファイルとソースファイルが保存されています。拡張子「.xcodeproj」のプロジェクトファイルをダブルクリックしてXcodeで開き、ツールバーの「Run」ボタンをクックすると、プログラムが作成され、実行されます。実行結果は、デバッグエリア（プロジェクトウインドウの中央の下側部分）に表示される「コンソールエリア」に出力されます。

実行結果が途切れる現象について

Xcode 4.3.2上でサンプルコードを実行した場合に、出力が最後まで行われない現象が起きることがあります。このような場合は、そのまま、再度、プログラムを実行してください。多くの場合、2回か3回の実行で最後まで表示されます。

CONTENTS

CHAPTER 01 Objective-Cの基礎知識

CHAPTER 02 Objective-Cの文法

CHAPTER 08 日付と時刻

CHAPTER 01

Objective-Cの基礎知識

Objective-Cとは

Objective-Cの概要

Objective-Cは、C言語を拡張し、オブジェクト指向言語として必要な機能を追加した言語です。その歴史は古く、1983年にまでさかのぼります。NeXTSTEPに採用され、現在では、Mac OS XやiPhone、iPadのネイティブアプリの標準的な開発言語になっています。

▶C言語がベース

C言語がベースとなっているので、多くの場合、C言語で書かれたコードはそのまま使用することができます。また、C言語で書かれた関数を直接、Objective-Cのコード内から呼び出すことができるので、C言語から利用できるライブラリやシステムAPIを直接、利用することができます。

▶C++とも組み合わせられる

Objective-CにさらにC++のコードを呼び出せるObjective-C++があります。Objective-C++では、いくつか例外はありますが、C++のコードをほぼそのまま使用できます。また、Objective-C++のコード内からC++のクラスなどを使用することもできます。

▶メソッドの引数にラベルを付けられる

メソッドの引数にはラベルを付けることができます。それにより、後から見てもプログラムコードをまるで普通の英文のようにとらえることができます。C言語やC++の場合、関数を呼び出しているところで引数が並んでいるのを見ただけでは、それが何を意味しているのかとらえにくいことがあります。しかし、Objective-Cでは、この引数のラベルによって、はじめて見たコードでも何を引数として指定しているのか、意味がとらえやすくなっています。

▶nilへのメッセージ送信

Objective-CのクラスのインスタンスでNULLのものを「nil」と表します。Objective-Cではnilインスタンスへのメッセージ送信（メソッド呼び出し）を行ってもクラッシュせず、何もしないという動作になります。このため、nilチェックを少なくできるため、コードがシンプルになります。シンプルなコードは見通しがよく、保守性が向上します。ただし、Objective-C++のコード内でも、C++のクラスのNULLのインスタンスのメソッドを呼び出すと、通常のC++と同じようにクラッシュしてしまうので、注意が必要です。

▶Automatic Reference Counting（ARC）とスマートポインタ

Objective-Cのクラスのインスタンスは、スマートポインタになっています。スマートポインタは参照カウンタを持っていて、確保された時点では参照カウンタは「1」になっています。参照カウンタが「0」になると、そのインスタンスは解放されます。複数の場所で1つのインスタンスが共有される場合などに、この参照カウンタを増減することで、インスタンスが使われている間は解放されないようにすることができます。

iOS 5.0以降、および、OS X 10.7以降（64ビットアプリケーション）向けでは、「Automatic

Referencing Counting」（以降、「ARC」と記述）を使用すると、参照カウンタの管理をコンパイラに任せることができます。iOS 4.x、および、OS X 10.6.xでも使用することができますが、一部制限があります。

▶ガベージコレクション

Objective-C 2.0以降では、ガベージコレクションを選択することもできるようになりました。ガベージコレクションでは参照カウンタはありません。使用されている間は解放されず、使用されなくなると自動的に解放されます。表面的な動作は「ARC」にも似ていますが、仕組みが大きく異なります。また、ガベージコレクションはiOSでは使用できないので、注意が必要です。

▶autorelease

ARCを使用していない、スマートポインタ方式のObjective-Cのクラスのインスタンスは、「autorelease」メソッドを呼び出すことで自動的に解放されるようにすることもできます。解放されるタイミングは、イベント処理がシステムに戻ったときです。正確には、そのインスタンスが登録された「NSAutoreleasePool」クラスのインスタンスが解放されるときに参照カウンタが1減ります。システムでは、イベント処理がシステムに戻ったときに解放される「NSAutoreleasePool」クラスのインスタンスを持っています。

ARCを使用している場合は、「NSAutoreleasePool」クラスは使用しません。「NSAutoreleasePool」クラスを使った仕組みではなく、コンパイラが必要なくなったと判断したタイミングで解放されるように、「release」メソッドの呼び出しを追加します。

このような仕組みによって、メソッドの戻り値としてクラスのインスタンスを返しても、呼び出し側が解放処理を忘れないかどうかを気にしないでよくなります。ARCを使用していない場合で、解放されては困るときだけ、呼び出し側で参照カウンタを増やして解放されないようにし、必要なくなったら参照カウンタを減らします。このとき、他から保持されておらず、参照カウンタが「0」になれば解放されます。

▶動的型付け

Objective-Cのインスタンスやクラスは、自身のクラスやクラスが持っているメソッド、親クラスの情報などを知っています。そのため、実行時に渡されたインスタンスのクラスは何か、このメソッドには対応しているかなどの型に関する情報を取得することができます。それだけではなく、インスタンスのクラスが何かは関係なく、特定のメソッドに対応していれば、そのメソッド呼び出すというようなこともできます。この特徴によって、非常に柔軟な設計ができます。

▶プロパティ

Objective-C 2.0以降ではプロパティを言語としてサポートしています。プロパティは条件を満たせば、必要なアクセッサメソッドを自動的に生成させることもできます。生成されるアクセッサメソッドは、排他制御が必要か、読み込み専用のプロパティかなど、柔軟に指定することができます。また、プロパティは、C言語の構造体のように「.」（ドット）演算子で読み書きしても、内部ではアクセッサメソッドが呼び出されるようになり、C言語の構造体のような手軽な読み書き方法で、柔軟なアクセス制御を行うことができます。

トールフリーブリッジ

Mac OS XやiOS用のプログラムを開発していると、場合によっては、Core Foundationを使用しなければいけないことがあります。Core Foundationでは、Objective-Cではなく、C言語のAPIを使用し、「CFStringRef」などの独自のオブジェクトを使用しなければいけません。

このようなときに力を発揮するのが、「トールフリーブリッジ」という仕組みです。トールフリーブリッジは、Objective-CのクラスとCore Foundationのオブジェクトを「キャスト」するだけで、相互利用できる仕組みです。たとえば、「NSString」クラスのインスタンスは「CFStringRef」のオブジェクトとして使用することができます。逆に、「CFStringRef」のオブジェクトは「NSString」クラスのインスタンスとして使用することができます。

機能豊富なフレームワーク

本書のメインターゲットである「Foundation」フレームワークを始め、機能豊富なフレームワークが標準で数多くそろえられています。「Foundation」フレームワークの機能を使用するだけでも、非常に多くのことができます。また、iOSやMac OS X用にはOSの機能を引き出すためのフレームワークが数多く用意されています。

また、標準で提供される「Foundation」フレームワークに「NSString」クラス（文字列）や「NSDictionary」クラス（辞書）、「NSArray」クラス（配列）などが用意されているので、複数のアプリで共通して使用するライブラリを開発するときにも、インターフェイスとしてこれらのクラスが利用できます。このようなクラスが標準では提供されない言語では、ライブラリごとに独自の配列クラスや構造体などを定義することになり、余分な開発工数がかかります。

そして、これらのフレームワークはOSの機能アップに追従して機能が今でも増えていっています。フレームワークの機能が増え、バイナリが新しくなるとバイナリの互換性が問題になりますが、Objective-Cの動的言語としての性質により、古いフレームワークで開発されたアプリとのバイナリの互換性を取りながら、メソッドの追加ができるようになっています。

開発環境について

開発環境の種類

Objective-Cでは、入力したプログラムコードをObjective-Cに対応したコンパイラを使用してコンパイルし、プログラムをビルドして実行するという流れになります。そのため、Objective-C対応のコンパイラが必要となります。Objective-C対応のコンパイラはMac OS Xでは、Xcodeをインストールすると同時にインストールされます。Mac OS X以外の環境でObjective-Cを使用したい場合には、GNUstepがあります。GNUstepはWindowsなどMac OS X以外のシステムにも対応しており、インストールすると同時にObjective-C対応のコンパイラや必要なライブラリも、一式、インストールされます。

Xcodeについて

XcodeはMac OS X標準の統合開発環境です。ソースコードを入力するためのテキストエディタ、ソースファイルをコンパイルしてプログラムをビルドするためのコンパイラおよびリンカ、プログラムをデバッグするためのデバッガなどの機能を持っています。正確には、Xcode自身はコンパイラやリンカのフロントエンドになっており、内部ではコマンドラインツールのコンパイラやリンカが動作するという形になっています。iOS用のネイティブアプリの開発にもXcodeを使用します。

▶Xcodeのセットアップ

Xcodeは、Mac App Storeからダウンロードすることができます。本書では、執筆時点で最新版のXcode 4.3.2を使用します。異なるバージョンをインストールする場合には、ファイル名や文字列が一部、異なる可能性があります。ご使用になる環境に合わせて読み替えてください。

Xcodeのインストールは、「App Store」アプリケーションを使用してMac App Storeからダウンロードします。本書の執筆時点では無償でダウンロードできますが、バージョンによっては有料の場合もあります。また、Xcode 4.3.2では、「アプリケーション」フォルダに「Xcode」がインストールされます。インストールされたXcodeを起動すると、初回起動時に追加コンポーネントのインストール画面が表示されるので、画面の指示に従ってインストールしてください。なお、少し前のバージョンでは、インストーラが「アプリケーション」フォルダにインストールされて、インストールされたインストーラを使用して、「Developer」フォルダにインストールされるようになっていました。

▶Xcodeのプロジェクトとターゲットについて

Xcodeでは、ビルドする対象ごとに「ターゲット」を作成します。「ターゲット」は、「プロジェクト」に登録されて管理されます。通常、1つのプログラムごとに「プロジェクト」を作成し、そのプログラムをビルドするために必要なプログラムやライブラリを「ターゲット」として同じプロジェクトに追加していきます。ただし、「ターゲット」の追加が必要なのは、そのプログラムやライブラリをソースコードからビルドするときです。すでにビルド済みのライブラリやフレームワークとリンクするだけのときには、「ターゲット」の追加は必要ありません。フレームワークやライブラリをビルドすることが

主体の「プロジェクト」では、テストプログラムなどを「ターゲット」として追加すると便利です。

このように、「プロジェクト」は、開発時に同時にビルドした方が効率がよいものや、同時にビルドしなければいけないものなどをひとまとめに管理することができます。なお、大きなプロジェクトになってくると、「プロジェクト」に「ターゲット」として登録するのでは都合が悪いケースも出てきます。このようなときには、「ワークスペース」を使用します。「ワークスペース」は、複数の「プロジェクト」をひとまとめにして管理できる仕組みです。「ターゲット」が「プロジェクト」に登録してまとめられるように、「プロジェクト」を「ワークスペース」に登録してまとめることができます。

▶ Xcodeのプロジェクトの作成

プロジェクトの作成は、次のようにして行います。

❶ 「File」メニューから「New」→「Project」コマンドを選択します。

❷ プロジェクトのテンプレートを選択するシートが表示されるので、テンプレートを選択し、「Next」ボタンをクリックします。本書で掲載しているサンプルコードはGUIを持たないコマンドラインツールで実行するので、シート左側のカテゴリから「Mac OS X」の「Application」を選択し、テンプレートから「Command Line Tool」を選択します。

❸ プロジェクトのオプションを設定するシートが表示されます。本書のサンプルを実行するためのプロジェクトの場合は、「Product Name」にプロジェクト名を入力し、「Company Identifier」には「com.yourcompany」と入力して、「Use Automatic Reference Counting」をONにし、「Type」から「Foundation」を選択して、「Next」ボタンをクリックします。

❹ プロジェクトの保存先を選択するシートが表示されるので、任意の場所を選択し、「Create」ボタンをクリックします。ローカルで、Gitを使ったバージョン管理を行いたい場合は、「Create local git repository for this project」をONにしてから「Create」ボタンをクリックします。

▶ プロジェクトへのファイルの追加

コンパイルするソースファイルやヘッダファイルは、プロジェクトに追加する必要があります。新規ファイルを作成して追加するときは、「File」メニューから「New」→「File」コマンドを選択します。既存のソースファイルを追加するときは、「File」メニューから「Add Files To "プロジェクト名"」コマンドを選択します。

▶ ビルドと実行

プロジェクトに登録したソースファイルのコンパイルおよびプログラムのビルドを行うには、「Product」メニューから「Run」コマンドを選択するか、ツールバーの「Run」ボタンをクリックします。エラーがある場合は、ナビゲータエリアにエラーがリストアップされます。リストアップされた項目を選択すると、編集エリアにソースファイルが表示され、エラー箇所までジャンプします。なお、ナビゲータエリアはプロジェクトに登録されたファイルの一覧も表示される領域で、ナビゲータエリアの上部に表示されたボタンで表示内容を切り替えることができます。

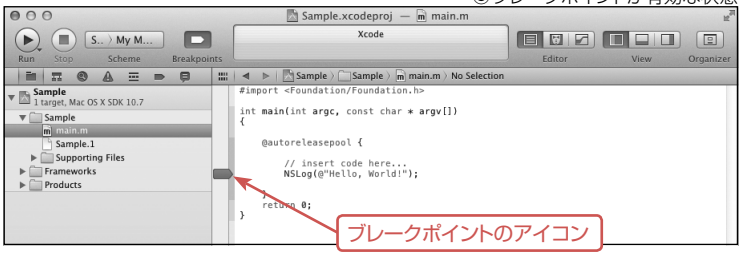
コマンドラインプログラムの場合は、実行結果はコンソールエリアに表示されます。コンソールエリアは編集エリアの下側に表示されます。プログラムがコンソールに出力すると表示されますが、「View」メニューから「Debug Area」→「Activate Console」コマンドを選択して表示することもできます。

▶デバッガの基本操作

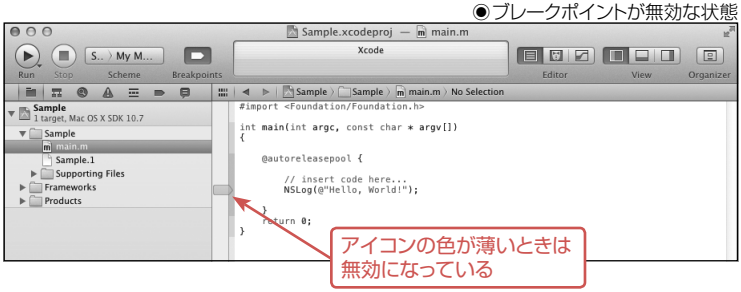
Xcodeはデバッガ機能を内蔵しています。プログラムを実行すると、編集エリアの下側に自動的にデバッグエリアが表示されます。コンソールもデバッグエリアの一部です。デバッグエリアには変数の値を表示するエリアもあり、一時停止したプログラムで、その瞬間の変数の値を確認することができます。デバッグエリアの上部には、プログラムの動作を制御するためのボタンが表示されています。一時停止したプログラムは、「Step over」ボタンをクリックすると1行だけ実行されます。「Step into」ボタンをクリックすると、実行しようとしているメソッド内や関数内に入ります。「Step out」ボタンをクリックすると、現在、実行中のメソッドや関数を呼び出しているコードに移動します。このとき、実行中のメソッドや関数の残りのコードも実行されます。

プログラムを任意の行で一時停止させるには、「ブレークポイント」を設定します。「ブレークポイント」の設定は、Xcodeでソースファイルを表示し、「ブレークポイント」を設定したい行の左側をクリックします。「ブレークポイント」のアイコンが表示されれば成功です。「ブレークポイント」を消去するときは、表示されている「ブレークポイント」のアイコンを、アイコンが表示される領域の外にドロップします。また、一時的に「ブレークポイント」を動かないようにしたいときは、「ブレークポイント」をクリックすると薄い表示になり、この状態のときは「ブレークポイント」がある行が実行されても一時停止しません。

◉ブレークポイントが有効な状態



◉ブレークポイントが無効な状態



GNUstepについて

GNUstepはCocoaやOpenStepとソースレベルで互換性がある開発環境です。ソースレベルでの互換性とは、同じソースファイルがCocoaとGNUstepのどちらの環境でも、おおむね修正なくコンパイルできるということです。GNUstepはオープンソースで提供され、WindowsやLinuxなど、さまざまなシステム上で動作します。GNUstepにはObjective-Cで開発を行うために必要なコンパイラやライブラリ、ヘッダファイルなどが含まれています。GNUstepは

GNUstep.orgからダウンロードすることができます。使用するときはライセンス条件をよく確認した上で、開発しようとしているプログラムで使用しても問題がないか検討をしてください。

▶ GNUstepのセットアップ

ここでは、Windows版のセットアップ方法を解説します。Windows版のGNUstepは、コンパイラやシェルなどのツールとライブラリファイル、ヘッダファイルで構成されています。GNUstep.orgから、次の4つのインストーラをダウンロードし、この順番でインストールしてください。

- GNUstep System
- GNUstep Core
- GNUstep Devel
- Cairo Backend

▶ ホームディレクトリ

標準設定でインストールを行うと、起動ドライブのルートディレクトリに「GNUstep」というフォルダが作られます。その中に「home」というフォルダが作られ、この中にユーザーごとに「ホームディレクトリ」が作成されます。「ホームディレクトリ」の作成は、はじめて「Shell」を起動したときに行われます。「Shell」は、「スタート」メニュー内に作成された「GNUstep」内のショートカットを選択して開きます。本書のサンプルコードを試すときには、「ホームディレクトリ」内にフォルダを作成し、その中にソースファイルやMakeファイルを作成してください。

▶ Makeファイルの作成

GNUstepを使ってビルドを行うには、ソースファイルと同じフォルダにMakeファイルを作成します。テキストエディタでソースファイルと同じフォルダに「GNUmakefile」という名前のファイル（拡張子は不要）を作成してください。ファイル内には、次のように入力します。

```
include $(GNUSTEP_MAKEFILES)/common.make

TOOL_NAME = TestTool              # 作成するプログラムの名前
TestTool_OBJC_FILES = main.m      # TOOL_NAMEで指定したプログラムで
                                  # 使用するソースファイルを指定する
                                  # 複数指定するときは半角スペースで
                                  # 区切る

include $(GNUSTEP_MAKEFILES)/tool.make
```

▶ ビルド

ビルドは「Shell」から行います。まず、「スタート」メニュー内に作成された「GNUstep」内の「Shell」を選択してShellを開きます。次に「cd」コマンドを使って「GNUmakefile」を作成したフォルダに移動し、「make」と入力します。「make」コマンドを実行すると、ソースファイルがコンパイルされ、プログラムがビルドされます。ビルドが成功すると「obj」フォルダが作られ、その中にプログラムが保存されているので、「Shell」でプログラム名を入力して実行してください。

関連項目 ▶▶▶

● コードの記述方法 …… p.41

コードの記述方法

ソースファイルの作成方法

コードを記述するソースファイルは、他のプログラミング言語と同様に、テキストエディタやXcodeなどの統合開発環境（IDE）で記述します。使い慣れているテキストエディタがあれば、それを使うこともできます。お勧めは統合開発環境です。Xcodeなどの統合開発環境は、コードを記述するだけではなく、ビルドやデバッグなども行えるので、効率よく開発を進めることができます。

拡張子

Objective-Cの拡張子は「.m」です。Objective-C++では「.mm」を使用します。また、C言語と同様にヘッダファイルも使います。ヘッダファイルの拡張子は「.h」です。

テキストエンコーディング

使用できるテキストエンコーディングは、コンパイラに依存します。Xcodeに付属するコンパイラでは、任意のテキストエンコーディングを使用することができます。WindowsやLinuxなど、他のプラットフォーム向けのプログラムと共有して使用しているソースファイルならば、そちらに合わせることもできます。なお、筆者の経験では、Mac OS X上ではUTF-8（BOMなし）が今までほとんどトラブルがなく、お勧めです。Windows版のプログラムと共有して使用するソースファイルなどで、シフトJISを使うこともありますが、ごくまれにコードが正しく認識されないことがありました。コードが正しいにもかかわらず、その通りに動作しないときなどにはテキストエンコーディングの変更を試してみるとよいと思います。

▶日本語について

UTF-8やシフトJIS、EUC-JPなど、日本語に対応したテキストエンコーディングを使用すれば、コード内のコメントに日本語を使用できます。しかし、文字列として、直接、日本語を渡すと、UTF-8以外では文字化けしてしまいます。プログラム内で日本語を使用したいときは、ローカライズ文字列の読み込みなど、実行時にファイルから読み込むようにしてください。

COLUMN　きれいなコード

プログラムコードを記述するとき、意識したいことにコードの「きれいさ」があります。きれいなコードは後から見てもわかりやすく、見通しがよくて保守しやすいコードになります。複雑な処理を書くときには汚くなりがちですので、意識的にきれいにできるところはきれいにしておきたいものです。プログラミング向けの機能を持ったテキストエディタならばオートインデントは当然として、他にもコードをきれいにするための整形機能が入っているものが数多くあります。

Objective-Cのコードを記述するとき、筆者はXcodeで記述しています。以前は

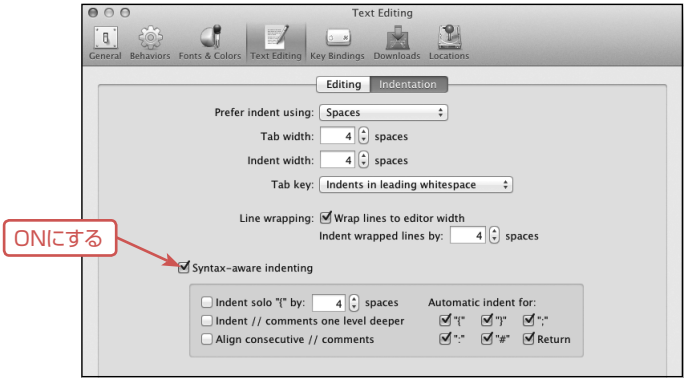
Xcodeはビルドとデバッグのみに使用して、コードの記述はプログラミング専用のテキストエディタを使っていました。しかし、現在はXcodeの「Syntax-aware indenting」という機能によるインデント処理が非常によく、すっかりXcodeで記述するようになりました。具体的には、次のようなObjective-Cのメソッドの構文に対応したインデント処理を知り、乗り換えました。

```
[anObject setParameter:param
             withOwner:self];
```

このコード自体には特に深い意味はなく、「anObject」というインスタンスの「setParameter:withOwner:」というメソッドを呼び出しているというコードです。このとき、Xcodeで記述すると、2行目の「:」を入力したときに、自動的に1行目の「setParameter」の後ろの「:」と2行目の「withOwner」の後ろの「:」を揃えるようにインデント処理が行われます。コードの記述中にXcodeがメソッドと認識できなかった場合などで「]」を入力した行しかインデント処理されなかったときも、後から中間の行の引数のラベルよりも前でタブを入力すると自動的に「:」を揃えるようにインデント処理を行ってくれます。

「Syntax-aware indenting」機能は、デフォルトで有効になっています。何らかの理由でOFFにしてしまった場合に、再度、有効にしたい場合は、次のように操作します。また、細かな挙動を好みに合わせて変更することもできます。

❶「Xcode」メニューから「Preferences」コマンドを選択します。
❷ 表示されたウインドウのツールバーの「Text Editing」アイコンをクリックします。
❸「Indentation」タブを表示します。
❹「Syntax-aware indenting」をONにします。
❺ 好みに合わせて「Syntax-aware indenting」の下側に表示されたオプションを変更します。
❻ ウインドウを閉じます。



関連項目 ▶▶▶

● 開発環境について …… p.37

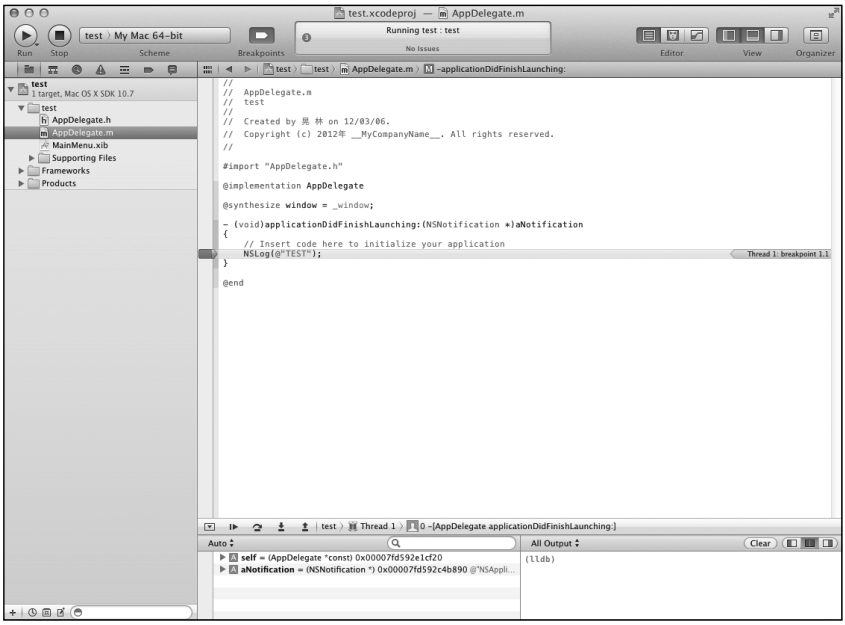
デバッグ時に便利な機能

デバッグ時に便利な機能

Xcodeが内部で使用するGDBやLLDBというデバッガには、デバッグ時に便利な機能があります。Foundationにも便利なログ出力関数があります。ここでは、知っているだけでデバッグ効率が大幅に変わってくる機能を紹介します。

▶デバッガの機能を呼び出す

Xcode上からはコンソールエリアを使用すると、内蔵デバッガに直接、コマンドを渡して実行することができます。Xcode 4では、内蔵デバッガとしてGDBとLLDBが用意されています。デフォルトでは、LLDBが使われるようになっています。デバッガに直接、コマンドを送るには、まず、ブレークポイントや一時停止ボタンを使用してプログラムを一時停止します。この状態で、デバッグエリアのコンソールエリアを表示すると、「(lldb)」と表示され、その後ろに入力できるようになっています。この状態でコマンドを入力すると、入力されたコマンドがデバッガに渡されます。コンソールエリアが表示されていない場合には、「View」メニューから「Debug Area」→「Activate Console」コマンドを選択すると表示されます。



▶Objective-Cのインスタンスの文字列表現を出力する

「po」というコマンドを使用すると、Objective-Cのインスタンスの文字列表現を出力することができます。文字列表現は「description」メソッドの戻り値です。たとえば、「NSString」クラスのインスタンスに対して行うと、インスタンスが持っている文字列が出力されます。「NSArray」

クラスのインスタンスでは配列に格納されている全インスタンスの「description」の戻り値が出力されるので、デバッグ中に配列の内容を調べることができます。独自に作成したクラスでも「description」メソッドを実装するだけで対応できるので、必要に応じて実装しておくとデバッグ中に便利です。

▶変数の内容を出力する

「int」などの変数の内容を出力したいときは、「print」コマンドを使用します。キャストも使えます。この関数が便利なのは、この後に紹介するメソッド呼び出しと組み合わせたときです。

▶デバッガ上でメソッドを実行する

デバッグ中に特に威力を発揮する機能の1つに、コンソールエリアでデバッグ中のプログラムのObjective-Cのメソッドを直接、呼び出せる機能があります。メソッドの戻り値がObjective-Cのインスタンスの場合は「po」コマンドと組み合わせ、それ以外の場合は「print」コマンドと組み合わせます。

たとえば、次のようなコードがあるとします。

```
- (NSArray *)doSomething:(int)value
{
    NSArray *newArray = [self createSomething:value];
    return newArray;
}

- (NSArray *)createSomething:(int)value
{
    NSMutableArray *newArray = [NSMutableArray arrayWithCapacity:0];

    if (value > 0)
    {
        for (int i = 0; i < value; i++)
        {
            [newArray addObject:[NSNumber numberWithInt:i]];
        }
    }
    else
    {
        for (int i = value; i < 0; i++)
        {
            [newArray addObject:[NSNumber numberWithInt:i]];
        }
    }
    return newArray;
}
```

このコードで、「return」の行（4行目）で一時停止しているとします。この状態で変数「value」に「10」が入っているときの「newArray」の内容を知りたいときは、次のように入力します。

```
po [self createSomething:10]
```

実行すると、「createSomething:」メソッドに「10」を渡してその結果が「po」コマンドによってコンソールエリアに出力されます。

```
(NSArray *) $1 = 0x0000000105914f80 <__NSArrayM 0x105914f80>(
0,
1,
2,
3,
4,
5,
6,
7,
8,
9
)
```

次に、変数「newArray」の要素数を知りたいとします。そのときは、次のように入力します。

```
print (int)[newArray count]
```

実行すると、「count」メソッドが実行され、戻り値が「int」型にキャストされてコンソールエリアに出力されます。

```
(int) $2 = 10
```

▶特定のメソッドで一時停止

「b」コマンドを使うと、特定の関数やメソッドで一時停止することもできます。次の例では、「NSLog」関数で一時停止するように設定しています。

```
b NSLog
```

入力すると次のように表示され、ブレークポイントが設定されます。

```
breakpoint set --name 'NSLog'
Breakpoint created: 2: name = 'NSLog', locations = 1, resolved = 1
```

表示された「Breakpoint」の後ろの「2」は「2番目のブレークポイント」という意味です。

次の例では、「NSMutableArray」クラスの「arrayWithCapacity:」メソッドで一時停止するように設定しています。

```
b +[NSMutableArray arrayWithCapacity:]
```

入力すると次のように表示され、ブレークポイントが設定されます。

```
breakpoint set --name '+[NSMutableArray arrayWithCapacity:]'
Breakpoint created: 3: name = '+[NSMutableArray arrayWithCapacity:]', locations = 1,
resolved = 1
```

なお、このようにして直接、LLDBに設定したブレークポイントは、一度、アプリを終了するとクリアされてしまいます。特定のメソッドに対するブレークポイントを覚えておくようにしたい場合には、「Product」メニューから「Debug」→「Create Symbolic Breakpoint」コマンドを使用してください。

▶「ブレークポイント」の一覧表示

設定されている「ブレークポイント」は、ナビゲータエリアのブレークポイントナビゲータに一覧表示されます。必要ないブレークポイントは、ここで削除することもできます。ブレークポイントナビゲータを表示するには、「View」メニューから「Navigators」→「Show Breakpoint Navigator」コマンドを選択すると表示されます。ナビゲータエリアの上部に表示された「Show the breakpoint navigator」ボタンをクリックしても表示できます。

COLUMN　中間ファイルの出力

デバッガ上でメソッドを実行することができる機能を利用して中間ファイルを出力すると、デバッグ時の作業効率が非常に上がります。たとえば、「NSDictionary」クラスの中身を知りたい場合、「po dict」などとすれば中身を表示することができますが、内容が複雑な場合や登録されているデータの個数が多い場合などは、この方法では作業効率が悪くなってしまいます。そこで、メソッドを実行する機能を使用して、プロパティリストファイルに書き出し、Xcodeやテキストエディタで開くようにします。たとえば、次のようにデバッガのコンソールに入力します。

```
print (int)[dict writeToFile:@"/data.plist" atomically:YES]
```

入力すると、変数「dict」が「/data.plist」にプロパティリストファイルとして書き出されます。

デバッグ中に内容がどんどん変わるときなども、ファイル名を変更しながら、書き出していき、Xcodeやテキストエディタ上で比較すると素早く値が変わったところを見つけることができ、作業効率が非常によくなります。

フレームワークについて

フレームワークについて

フレームワークは、Mac OS Xにおいて、ライブラリのオブジェクトコード、ヘッダファイル、フレームワークの情報ファイル、その他の必要なリソースファイルをひとまとめにしたフォルダです。Mac OS Xではこのようなフォルダをバンドルと呼びます。バンドル内には決められたルールがあり、それに従ってオブジェクトコードやヘッダファイルが格納されます。また、バンドルのフォルダには、拡張子が付けられます。Mac OS Xではアプリケーションも1つのバンドルですが、Finderからはフォルダではなく、1つのファイルとして表示されます。このようなバンドルをパッケージと呼びます。

Objective-Cの文字列やコレクションなどのクラスは、「Foundation」フレームワークによって提供されています。この「Foundation」フレームワークと、その他、Mac OS Xのシステムの APIを提供するフレームワークをあわせて「Cocoa」と呼ばれています。iOSでも同様に「Foundation」フレームワークが提供され、その他、iOSのシステムのAPIを提供するフレームワークを合わせて「Cocoa Touch」と呼ばれています。

「Foundation」フレームワーク以外の代表的なフレームワークには、「AppKit」フレームワークがあります。「AppKit」フレームワークは、メニューバーやウインドウ、ボタンなどのコントロールなど、ユーザーインターフェイスに関する機能を提供しています。画像に関する機能や描画処理なども「AppKit」フレームワークが提供しています。「Cocoa Touch」では「AppKit」フレームワークは提供されません。代わりに「UIKit」フレームワークが提供されています。

なお、バンドルおよびパッケージの拡張子は、右の表のようになります。

バンドル・パッケージ	拡張子
アプリケーション	.app
フレームワーク	.framework
プラグイン	.plugin
カーネル拡張	.kext
汎用的なバンドル	.bundle

▶フレームワークの配置されるディレクトリ

フレームワークは「Frameworks」というディレクトリに配置されます。「Frameworks」ディレクトリは複数の場所にあり、その利用方法によって配置する場所を変更します。

◉「Frameworks」ディレクトリの場所と利用方法

ディレクトリパス	目的
/System/Library/Frameworks	システム全体で使用するフレームワークを配置する。このディレクトリはOSによって重要なフレームワークが配置されるので、ユーザーアプリケーションのフレームワークは配置するべきではない
/Library/Frameworks	システム全体で使用するフレームワークを配置する。ユーザーアプリケーションでユーザーアカウントに関係なく使用するフレームワークを配置する
~/Library/Frameworks	ユーザー専用のフレームワークを配置する。「~/」はログインしているユーザーのホームディレクトリ
*.app/Contents/Frameworks	特定のアプリケーション専用のフレームワークを配置する

インスタンスの確保と解放

インスタンスの確保と解放について

Objective-Cでは、クラスのインスタンスの確保は「alloc」メソッドを使用します。確保したインスタンスは、「init」メソッドで初期化を行います。各クラスは「init」メソッドを定義して、その中でインスタンスの初期化を行います。「init」メソッドは、イニシャライザと呼ばれています。イニシャライザはクラスによって引数が異なり、メソッド名が異なるものも定義されます。ただし、いずれも「init」から始まります。

ARCを使用しない場合は、確保したインスタンスは「release」メソッドで解放する必要があります。「release」メソッドが呼ばれると、そのインスタンスの参照カウンタが1減り、参照カウンタが「0」になると実際の解放処理が行われます。このとき、「dealloc」メソッドが呼び出されます。各クラスは「dealloc」メソッドを定義して、その中でインスタンスの解放処理を実装します。

▶イニシャライザの戻り値について

イニシャライザの戻り値は、親クラスのイニシャライザの戻り値を返すようにします。この戻り値は、インスタンス自身です。イニシャライザは失敗することもあります。そのときには、インスタンスを解放して「nil」を返します。

```
- (id)init
{
    self = [super init];
    if (self)   // 親クラスの初期化処理が失敗したら、「nil」が返される
    {
        // このクラスの初期化処理
    }
    return self;
}
```

▶「dealloc」メソッドは親クラスの呼び出しも忘れずに

「dealloc」メソッドを実装した場合で、ARCを使用していない場合には、親クラスの「dealloc」メソッドの呼び出しを忘れないように注意してください。忘れてしまうと親クラスの解放処理が行われず、メモリリークや予期しない動作の原因になります。逆に、ARCを使用している場合は、親クラスの「dealloc」メソッドを呼び出すことはできないので書かないようにします。ARCを使用している場合は、親クラスの「dealloc」メソッドは自動的に呼ばれます。

```
- (void)dealloc
{
    // このクラスの解放処理
    // ...
    [super dealloc];  // ARCを使用していないときのみ親クラスの解放処理を呼び出す
}
```

▶「autorelease」メソッドについて

ARCを使用していないコードでのインスタンスの解放には、「release」メソッドによる明示的な解放の他に、「autorelease」メソッドによる自動解放があります。ARCを使用している場合は、自動的に解放処理が追加されるので、「autorelease」メソッドを呼び出すことはできません。「release」メソッドの代わりに「autorelease」メソッドが呼ばれると、そのインスタンスは「NSAutoreleasePool」クラスのインスタンスが解放されるときに同時に解放されるようになります。正確には「release」メソッドと同様に、そのタイミングで参照カウンタが1減ります。システムもイベントを回すたびに「NSAutoreleasePool」クラスのインスタンスを作成しているので、システムに処理が戻ると「NSAutoreleasePool」クラスのインスタンスが解放され、「autorelease」メソッドを呼び出したメソッドが解放されます。

「NSAutoreleasePool」クラスは、「autorelease」メソッドが呼ばれたインスタンスを管理するクラスです。「autorelease」メソッドを呼び出すと、有効な「NSAutoreleasePool」クラスのインスタンスの中で最後に作られたインスタンスに登録されます。

```
NSAutoreleasePool *pool1 = [[NSAutoreleasePool alloc] init];

while (...)
{
    NSAutoreleasePool *pool2 = [[NSAutoreleasePool alloc] init];

    // このインスタンスは「pool2」に登録される
    MyObject *obj = [[[MyObject alloc] init] autorelease];

    // この「drain」メソッドで「obj」は解放される
    [pool2 drain];
}

// このインスタンスは「pool1」に登録される
MyObject *obj2 = [[[MyObject alloc] init] autorelease];

// この「drain」メソッドで「obj2」は解放される
[pool1 drain];
```

「drain」メソッドは、「NSAutoreleasePool」クラスのインスタンスを解放するメソッドです。他のメソッドと同様に「release」メソッドを使用することもできますが、ガベージコレクション環境との互換性のために「NSAutoreleasePool」クラスでは「drain」メソッドを使用します。なお、「drain」メソッドが使用できるのは、Mac OS X 10.4以降、もしくは、iOSです。Mac OS X 10.4未満では「release」メソッドを使用してください。

本書では、「@autoreleasepool」文を使用するので「NSAutoreleasePool」クラスは基本的には使用しませんが、使用する場合は「drain」メソッドで記述します。Mac OS X 10.4未満では、「release」メソッドに置き換えてください。

ARCを使用している場合には内部では自動解放プールは使用されていますが、直接、

アプリ側から「NSAutoreleasePool」クラスを使用することはできません。代わりに「@autoreleasepool」構文を使用して、次のように記述します。

```
while (...)
{
    @autoreleasepool
    {
        // ... 自動解放されるオブジェクトが作成されるコード
    }
}
```

なお、「@autoreleasepool」文は、ARCを使用しないコードでも記述することができます。

▶「autorelease」メソッドを使うべき場面

メソッドの戻り値など、「alloc」メソッドを呼び出したメソッドが終了した後もそのインスタンスが存在するならば、「autorelease」メソッドを呼び出して返すべきです。そのようにすることで、そのインスタンスを受け取った側の処理は、インスタンスの解放義務がなくなります。また、呼び出し側の処理も、「alloc」メソッドによる明示的なインスタンス確保以外で取得したインスタンスは自動解放されると想定しています（ただし、例外もある）。

「Foundation」フレームワークも同様に、「alloc」メソッドによる明示的な確保以外でのインスタンスの確保では、「autorelease」メソッドが呼ばれたインスタンスが返されます。

▶「copy」メソッドと「mutableCopy」メソッドは「autorelease」メソッドが呼ばれていない

「alloc」メソッド以外に、「copy」メソッドと「mutableCopy」メソッドで確保したインスタンスも「autorelease」メソッドが呼ばれていないので注意してください。これらのメソッドで取得したインスタンスは、ARCを使用していないコードでは、「release」メソッドで明示的に解放するか、「autorelease」メソッドを呼び出す必要があります。

▶インスタンスを保持するときは「retain」メソッドを呼ぶ

メソッド呼び出しによって取得したインスタンスは、「autorelease」メソッドが呼ばれている場合がほとんどです。ARCを使用していないコードで、取得したインスタンスをクラスのインスタンス変数やグローバル変数に代入して保持するときは、必ず「retain」メソッドで参照カウンタを増やすようにします。これを忘れてしまうと、イベントループの処理がシステムに戻ったときに解放されてしまい、その後、その保持していたインスタンスを使用すると、すでに解放されているのでクラッシュしてしまいます。また、「retain」メソッドで参照カウンタを増やしたインスタンスが必要なくなったら、「release」メソッドで参照カウンタを減らします。このとき、他から参照されておらず、参照カウンタが「0」になれば解放されます。

ARCを使用している場合は、「retain」メソッドを呼ぶことはできないので、プロパティに格納して保持するようにします。このとき、他のオブジェクトが保持していないオブジェクトの場合は、強い参照関係を持つプロパティに保持するようにします。

▶ガベージコレクションが有効なとき

ガベージコレクションが有効なときは、「release」メソッドと「autorelease」メソッドは何もしま

せん。インスタンスの解放は、どこからも使用されなくなったときにシステム側が自動的に行うからです。また、「dealloc」メソッドも呼ばれません。代わりに「finalize」メソッドが呼ばれます。

しかし、「finalize」メソッドは「dealloc」メソッドとは異なり、どのタイミングで呼ばれるかはガベージコレクション次第なので、順序やタイミングに左右されるようなコードを書かないようにします。また、「finalize」メソッド内の処理はスレッドセーフにしなければいけません。なるべく、「finalize」メソッドに頼らない設計にするべきです。

▶ ARCについて

ARCは、iOS 5.0以降、もしくは、OS X 10.7以降で動作する64ビットアプリケーションで使用可能な新しいメモリ管理機能です。表面的にはガベージコレクションのようにも見えますが、実際には、メモリ管理モデルは参照カウンタ方式で、参照カウンタを増減させる処理をコンパイラが自動的に挿入するという方式になっています。そのため、アプリ側で明示的に、「release」メソッドや「autorelease」メソッド、「retain」メソッドなどの参照カウンタを変更するためのメソッドは呼び出すことができないようになっています。

必要なくなると自動的に解放されるので開発者の負担が減りますが、一方で、使用しないコードとのソースコードレベルでの互換性がとりにくいのが難点です。たとえば、ARCを前提にしたコードをそのままARCを使用しないプロジェクトに持って行くと、解放処理が書かれていないのでメモリリークになってしまいます。しかし、単純な解放漏れを防止できるなどのよい面もあるので、開発対象のアプリの動作環境によって使用可能ならば、利用した方がよいと思います。本書では、ARCを使用している設定でのサンプルコードを掲載していますが、ARCを使用しない場合に記述する必要があるコードはコメントで記述しています。

関連項目 ▶▶▶
● プロパティ定義について ……p.106

Objective-Cのクラスについて

Objective-Cのクラスの特徴

Objective-Cのクラスも、他のオブジェクト指向言語と同じような特徴を持っています。クラスは派生することができ、親クラスと子クラスが存在します。子クラスは親クラスのメソッドをオーバーライドすることもできますし、子クラスは親クラスの処理を呼び出すこともできます。その他、次のような特徴があります。

- 多重継承はできない
- ほとんどのクラスは「NSObject」クラスを継承している
- 動的型付け
- メソッド名とインスタンス変数名は重複してもよい

▶多重継承はできない

C++とは異なり、Objective-Cはクラスの多重継承はできません。しかし、「プロトコル」があるので、多重継承とは違った形でインターフェイスの定義が行えます。「プロトコル」は、特定のメソッドがあることを定義します。これはクラスが何であるかは関係なく、特定のメソッドに対応していればよいという考え方です。

わかりやすい例では、「AppKit」フレームワークの「NSTableView」クラスと「NSTableDataSource」プロトコルの関係です。「NSTableView」クラスは、表示するべき情報を提供するデータソースとなるインスタンスを必要とします。このデータソースは「NSTableDataSource」プロトコルで定義されるメソッドを実装したクラスであれば、クラスの種類は問いません。「NSTableView」が情報を表示するのに必要とするのは、情報そのものと、その情報を取得するためのメソッドだけです。つまり、特定のクラスから派生したクラスである必要はありません。「NSTableView」クラスは、データソースとして設定されたインスタンスに対して、「NSTableDataSource」プロトコルで定義した、情報を取得するためのメソッドを呼び出します。そして、取得した情報を表示します。

▶ほとんどのクラスは「NSObject」クラスを継承している

「NSObject」クラスは、非常に重要な基本クラスです。Objective-Cのほとんどのクラスは「NSObject」クラスを継承しています。

「NSObject」クラスは、インスタンス確保に使用する「alloc」メソッドや、解放に使用する「release」メソッド、参照カウンタを増やす「retain」メソッド、自動解放されるようにするための「autorelease」メソッドなど、非常に重要なメソッドを実装しています。新しいクラスを作成するときは、必ず「NSObject」クラスか、その他のクラスのサブクラスとして作成してください。

▶メソッドについての注意点

Objective-Cのクラスは、クラス自身が自身の情報を知っています。メソッドがあるかないかは実行時に解決されます。そのため、C++ではメソッド名が間違っていればビルドが失敗しますが、Objective-Cではビルドは成功します（ただし、警告が出る）。しかし、クラスが持っていないメソッドを呼び出そうとすれば、実行時に例外が投げられ、そこで処理が中断されます。ビルド時にはコンパイラが出力する警告にも注意してください。

C言語のコードとの組み合わせ

C言語のコードと組み合わせる

Objective-CはC言語を拡張した言語なので、C言語のコードはそのまま使用でき、お互いの機能を組み合わせることもできます。

▶関数

C言語の関数内でObjective-Cのメソッド呼び出しができます。逆にObjective-Cのメソッド内でC言語の関数を呼び出すこともできます。これらの特徴により、C言語用のライブラリのコールバック関数をC言語のコードとして作成し、その中でObjective-Cの機能を使うことや、Objective-Cのコード内から外部のC言語用のライブラリを呼び出すことができます。また、C言語で書かれたコードを書き直さなくても、Objective-Cのクラスでラップするだけでも既存のC言語のコードが扱いやすくなります。

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

@interface MyObject : NSObject
{
}
- (void)doSomething:(NSString*)str;
@end

@implementation MyObject

- (void)doSomething:(NSString*)str
{
    // Objective-Cのメソッド内でC言語の関数を呼び出せる
    printf("%s", [str UTF8String]);
}

@end

void doSomething(void)
{
    // C言語の関数内でObjective-Cのクラスのインスタンスを作成できる
    NSString* str = [NSString stringWithString:@"TEST"];
    // C言語の関数内でObjective-Cのクラスのメソッドを呼び出せる
    str = [str stringByAppendingString:@" STRING"];
    // C言語の関数を呼び出せる
    printf("%s\n", [str UTF8String]);
}

int main (int argc, const char * argv[])
```

```
{
    @autoreleasepool
    {
        MyObject* myObject = [[MyObject alloc] init];
        [myObject doSomething:@"Objective-C method\n"];

        doSomething();

        // ARCを使用しない場合は、次の行で解放する
        // [myObject release];
    }
    return 0;
}
```

このコードを実行すると、次のように表示されます。

```
Objective-C method
TEST STRING
```

▶構造体

構造体もそのまま使用できます。メソッドの引数やクラスのインスタンス変数などにも使うことができます。Objective-Cのクラス宣言内での構造体の宣言もできます。ただし、Objective-Cにはネームスペースの機能はないので、クラス内に宣言してもグローバルネームスペースに配置されます。そのため、宣言した構造体は、その宣言を持つObjective-Cのクラスに属するのではなく、自由に他のクラスや関数から使用できます。これは、C++とは異なる点になるので、注意してください。

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

@interface MyObject : NSObject
{
    // Objective-Cのクラス宣言内で構造体を宣言できる
    struct MyStruct {
        int     data0;
        int     data1;
    } MyStruct;

    // 構造体の変数もObjective-Cクラス宣言内で使用できる
    struct MyStruct     _data;
}
@end

@implementation MyObject

- (id)init
```

```
{
    self = [super init];
    if (self)
    {
        _data.data0 = 3;
        _data.data1 = 2;
    }
    return self;
}

- (void)printData
{
    NSLog(@"%d, %d", _data.data0, _data.data1);
}

- (void)printDataFromExternal:(struct MyStruct)externalData
{
    NSLog(@"%d, %d", externalData.data0, externalData.data1);
}

@end

int main (int argc, const char * argv[])
{
    @autoreleasepool
    {
        MyObject* myObject = [[MyObject alloc] init];
        [myObject printData];

        // Objective-Cのクラス宣言内で宣言した構造体はグローバル扱い
        struct MyStruct data = { 10, 20 };
        [myObject printDataFromExternal:data];

        // ARCを使用しない場合は、次の行で解放する
        // [myObject release];
    }
    return 0;
}
```

このコードを実行すると、次のように表示されます。

```
2012-03-09 23:06:29.841 CStruct[452:403] 3, 2
2012-03-09 23:06:29.846 CStruct[452:403] 10, 20
```

▶型定義

型定義を行う「typedef」命令もそのまま使用可能です。構造体と同様にメソッドの引数やクラスのインスタンス変数などにも使うことができます。

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

// 整数型を型定義する
typedef unsigned long long  SerialNumber;

// 構造体を型定義する
typedef struct {
    SerialNumber    sn;
    int             data0;
    int             data1;
} MyStruct, *MyStructPtr;

int main (int argc, const char * argv[])
{
    @autoreleasepool
    {
        MyStructPtr p = (MyStructPtr)malloc(sizeof(MyStruct));

        p->sn = 10;
        p->data0 = 20;
        p->data1 = 30;

        NSLog(@"sn=%llu, data0=%d, data1=%d",
              p->sn, p->data0, p->data1);

        free(p);
    }
    return 0;
}
```

このコードを実行すると、次のように表示されます。

```
2012-03-09 23:09:39.523 Typedef[480:403] sn=10, data0=20, data1=30
```

C++のクラスについて

Objective-C++でのC++のクラスの扱いについて

Objective-C++では、C++のクラスを使うことができます。C++のクラスのインスタンス変数やメソッドの引数に、Objective-Cのインスタンスを渡すことができます。逆にObjective-Cのクラスのインスタンス変数やメソッドの引数に、C++のクラスのインスタンスやインスタンスポインタを使用することもできます。C++のクラスのインスタンスはインスタンスポインタとスタック上に確保したインスタンスがありますが、仮想関数やコンストラクタ、デストラクタにはObjecitve-C++は対応していないので、Objective-Cのクラスのインスタンス変数としては、インスタンスポインタを持つ方が無難です。

また、クラス階層の混在はできません。C++のクラスを継承したObjective-Cのクラスを作成することや、逆にObjective-Cのクラスを継承したC++のクラスを作成することもできません。

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

class MyClass {
public:
    MyClass() {
        _str = nil;
    }

    virtual ~MyClass() {
        // ARCを使用しない場合は、次の行で解放する
        // [_str release];
    }

    // Objective-Cのクラスのインスタンスを C++のメソッドの引数として使える
    void DoSomething(NSString *newStr) {
        if (_str != newStr)
        {
            // ARCを使用しない場合は、次の行で解放する
            // [_str release];
            _str = [newStr copy];
        }
        NSLog(@"%@", _str);
    }

private:
    // Objective-Cのクラスのインスタンスをインスタンス変数として使用できる
    NSString *_str;
};
```

```
@interface MyObject : NSObject
{
    // C++のクラスのインスタンスポインタをObjective-Cのクラスの
    // インスタンス変数として使用できる
    MyClass *_data;
}
// Objective-Cのメソッドの引数にC++のインスタンスポインタを使用できる
- (void)doSomething:(MyClass *)p;
@end

@implementation MyObject

- (void)doSomething:(MyClass *)p
{
    _data = p;
    if (_data)
    {
        _data->DoSomething(@"Call From Objective-C Method");
    }
}

@end

int main(int argc, char **argv)
{
    @autoreleasepool
    {
        MyClass *p = new MyClass();
        MyObject *p2 = [[MyObject alloc] init];

        [p2 doSomething:p];
        delete p;

        // ARCを使用しない場合は、次の行で解放する
        // [p2 release];
    }
    return 0;
}
```

このコードを実行すると、次のように表示されます。

```
2012-03-09 23:21:40.784 CppClass[573:403] Call From Objective-C Method
```

仮想関数には対応していない

Objective-C++は、C++の仮想関数には対応していません。そのため、Objective-Cのクラスのインスタンス変数としてvirtualメンバーを持つC++のクラスのインスタンスを置くことはで

きません。しかし、完全に対応していないというわけではなく、Objective-Cのクラスのインスタンス変数としてC++のクラスを使いたいときは、「new」で確保したインスタンスポインタを持つようにします。インスタンスポインタの場合は仮想関数を持つクラスも使用することができ、「->」演算子を用いたメソッド呼び出しでは仮想関数も呼び出すことができます。これはインスタンス変数だけではなく、スタック変数やメソッドの引数でも同様です。

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

class MyClass {
public:
    MyClass() { NSLog(@"MyClass::MyClass()"); }
    virtual ~MyClass() { NSLog(@"MyClass::~MyClass"); }
};

class MyClass2 {
public:
    MyClass2() { NSLog(@"MyClass2::MyClass2()"); }
    ~MyClass2() { NSLog(@"MyClass2::~MyClass2()"); }
};

@interface MyObject : NSObject
{
    /*
     MyClass     _myClass;       // 仮想関数を持っているので、
     // インスタンス変数にはできない
     // (エラーになる)
     */
    MyClass     *_myClassPtr;   // 問題なし
    MyClass2    *_myClass2Ptr;  // 問題なし
}
- (void)doSomething;
@end

@implementation MyObject

- (void)doSomething
{
    // 「new」がコンストラクタを呼び出す
    _myClassPtr = new MyClass();
    // 「delete」がデストラクタを呼び出す
    delete _myClassPtr;

    // 「new」がコンストラクタを呼び出す
    _myClass2Ptr = new MyClass2();
    // 「delete」がデストラクタを呼び出す
    delete _myClass2Ptr;
```

```
    // メソッド内でスタック内にC++のクラスを置くことは可能
    // ここでコンストラクタが呼ばれる
    MyClass c1;
    // メソッドを抜けるときに、変数「c1」のデストラクタが呼ばれる
}

@end

int main(int argc, char **argv)
{
    @autoreleasepool
    {
        MyObject *myObject = [[MyObject alloc] init];

        [myObject doSomething];

        // ARCを使用しない場合は、次の行で解放する
        // [myObject release];
    }
    return 0;
}
```

このコードを実行すると、次のようになります。なお、日付と時刻は実行時の日時です。

```
2012-03-09 23:25:20.810 CppClass2[619:403] MyClass::MyClass()
2012-03-09 23:25:20.814 CppClass2[619:403] MyClass::~MyClass
2012-03-09 23:25:20.815 CppClass2[619:403] MyClass2::MyClass2()
2012-03-09 23:25:20.816 CppClass2[619:403] MyClass2::~MyClass2()
2012-03-09 23:25:20.817 CppClass2[619:403] MyClass::MyClass()
2012-03-09 23:25:20.818 CppClass2[619:403] MyClass::~MyClass
```

Objective-C++内ではC++のコンストラクタもデストラクタも呼び出されます。

COLUMN　Objective-Cからも読み込まれるヘッダファイル内でのC++のクラスの扱い

通常、Objective-Cのクラスの定義はヘッダファイルで行いますが、ヘッダファイルで定義しているクラス内でC++のクラスのインスタンスポインタをインスタンス変数として使用すると、そのヘッダファイルはObjective-Cのファイル（拡張子が「.m」のファイル）では読み込むことができなくなります。これは、Objective-C++はC++のクラスを理解することができますが、Objective-CはC++のクラスを理解することができないためです。このようなケースが発生する場合は、Objective-Cからは「void *」として読み込ませ、Objective-C++からは本来のクラスのインスタンスポインタになるように、「typedef」文を使用します。

たとえば、「MyClass」というC++のクラスがあるとして、これを「MyObject」というObjective-Cのクラスのインスタンス変数で持つ場合は、次のようにします。まず、ヘッダ

ファイルでは、次のように記述します。

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

#ifdef __cplusplus

// C++とObjective-C++用の定義
class MyClass;
typedef MyClass* MyClassPtr;

#else

// Objective-C用の定義
typedef void* MyClassPtr;

#endif

@interface MyObject : NSObject
{
    MyClassPtr  _myClassPtr;
}

- (void)doSomething;

@end
```

ここで、定義した「MyClassPtr」という型は、Objective-Cからは「void ＊」として定義され、C++、および、Objective-C++からは「MyClass ＊」として定義されます。これによって、「MyObject」クラスの実装ファイル（拡張子が「.mm」のファイル）では、次のように記述できます。

```
#import "MyObject.h"
#import "MyClass.h"

@implementation MyObject

- (id)init
{
    self = [super init];
    if (self)
    {
        // C++のクラスのインスタンスを作成する
        _myClassPtr = new MyClass();
    }
    return self;
}
```

```
- (void)dealloc
{
    // C++のクラスのインスタンスを破棄する
    if (_myClassPtr)
        delete _myClassPtr;

    // ARCを使用しない場合は、次の行で親クラスのメソッドを呼び出す
    // [super dealloc];
}

- (void)doSomething
{
    // C++のクラスのメソッドを呼び出す
    if (_myClassPtr)
        _myClassPtr->DoSomething();
}

@end
```

「MyClass.h」ファイルは、C++のクラス定義です。たとえば、次のようなコードにします。

```
#ifndef CppClass3_MyClass_h
#define CppClass3_MyClass_h

class MyClass
{
public:
    MyClass() {}
    virtual ~MyClass() {}

    void DoSomething()
    {
        printf("Called from C++\n");
    }
};

#endif
```

このコードを実行すると、次のように表示されます。

```
Called from C++
```

ネームスペースについて

Objective-C++でのネームスペースの扱いについて

Objective-C++では、C++のクラスに対してはネームスペースを使うことができます。制限事項はありますが、1つのコード内でネームスペースがあるC++のコードとObjective-Cのコードとを組み合わせることができます。「using namespace」命令を使ったネームスペースの省略も使用できます。

ただし、Objective-C自体にはネームスペースという概念はないので、Objective-Cのクラス名が衝突してしまったときには、避ける手段がありません。クラス名を変更する以外に方法がないので、1つのプロジェクト内や1つのプログラム内で共通して使用するプレフィックス文字列を決めて、それをクラス名の先頭に付けるなど、クラス名が重複しないようにする必要があります。たとえば、Objective-CのFoundationフレームワークや「AppKit」フレームワークでは、「NS」から始まるクラス名を使用しています。

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

// Objective-Cのクラス宣言の外でのネームスペースの定義は可能
namespace MyNamespace2 {
    // C++のクラスなのでネームスペース内で宣言できる
    class MyClass {
    public:
        MyClass() {}
        virtual ~MyClass() {}
        void Do() { NSLog(@"MyClass::Do()"); }
    };
}

int main (int argc, const char* argv[])
{
    @autoreleasepool
    {
        // 関数内でネームスペース宣言内のC++のクラスを使うことも可能
        MyNamespace2::MyClass *p = new MyNamespace2::MyClass();
        p->Do();
        delete p;
    }
    return 0;
}
```

▶ Objective-C++でのネームスペースの制限事項

Objective-C++でのC++のネームスペースについては、次の2つの制限事項があります。

- Objective-Cのクラス宣言内ではネームスペースを宣言できない
- ネームスペース内ではObjective-Cのクラスを宣言できない

この2つの制限事項があるため、次のようなコードはエラーになります。

```
@interface MyObject : NSObject
{
    // Objective-Cのクラス宣言内ではネームスペースは宣言できない(エラーになる)
    namespace MyNamespace
    {
    }
}
@end

namespace MyNamespace2
{
    // Objective-CのクラスをC++のネームスペース内で宣言することはできない(エラーになる)
    @interface MyObject2 : NSObject
    {
    }
    @end
}
```

CHAPTER 02

Objective-Cの文法

リテラルについて

Objective-Cのリテラル

リテラルは、プログラムコード中に、直接、記述できる、文字や文字列、数値などです。Objective-Cのリテラルは C言語から引き継いだものです。

▶数値

数値は10進数、16進数、8進数が使用できます。また、整数だけではなく実数も使用できます。普通に数値を書くと10進数として扱われます。「0x」で始めると16進数、「0」から始めると8進数として扱われます。また、数値の後ろに付けるサフィックスによって何の型として扱われるかも変わります。特に「long long」型でしか表現できないような大きな数値の場合、「ll」を付けないと数値が丸められて正しい動作をしないこともあるので、注意してください。

```
20          // 整数
-14         //「-」を付けると負の整数
10.4        //「.」を付けると実数
-11.5       //「-」と「.」を両方とも使えば負の実数
20.345f     //「f」を付けると「float」型の扱い
403u        //「u」を付けると符号なしの扱い
20005ul     //「ul」を付けると「unsigned long」型の扱い
205l        //「l」を付けると「long」型の扱い
12345ll     //「ll」を付けると「long long」型の扱い
20345ull    //「ull」を付けると「unsigned long long」型の扱い
0x1f        //「0x」を付けると16進数となる
0x1F        // 16進数のアルファベットは大文字・小文字のどちらも使える
0214        //「0」から始めると8進数となる
```

▶BOOL型

「真」か「偽」を表します。値は「YES」か「NO」です。「YES」は1と等価で、「NO」は「0」と等価です。

▶文字

文字は「'」記号で囲んで表現します。型は「char」型として扱われます。プレフィックスに「L」を付けると、「wchar_t」型として扱われます。アルファベットの大文字と小文字は、別々の値になるので注意してください。また、「数字」と「数値」も異なるので注意してください。

```
// 大文字の「A」と小文字の「a」は別の文字として扱われる
'A'
'a'
// 数値と文字としての「1」は別の値となる
'1'
// 「wchar_t」型として扱われる
L'A'
```

▶文字列

文字列は「"」記号で囲んで表現します。C言語と同様に「NULL終端文字列」となります。「NULL終端文字列」とは、文字列の最後に文字列の終端を表す文字として「0」('0')が入っている文字列です。文字列は「char」型の変数が連続して並んだ、「char」型の配列と同じです。「文字」と同様にプレフィックスに「L」を付けると、「wchar_t」型の変数が連続して並んだ、「wchar_t」型の配列と同じになります。ここで「同じ」という曖昧な表現になっているのは、実際には「char*」ないし「wchar_t*」となるからです。「*」は「ポインタ」を表現します。

それに加えて、Objective-Cでは、プレフィックスに「@」を付けると「NSString」クラスのインスタンスとして扱われます。詳しくは、《**文字列について**》(p.160)を参照してください。

```
// 「char*」として扱われる
"This is a NULL terminated string"
// 「wchar_t*」として扱われる
L"This is a NULL terminated wide string"
// 「NSString*」として扱われる
@"This is a NSString*."
```

COLUMN **文字の実体は数値**

文字の実体は数値です。各文字の値がいくつになるかはテキストエンコーディング(文字コード)により定義されています。多くのテキストエンコーディングでは127以下の値はASCIIコードと互換性を保っているので、アルファベットや数字は直接、ソースコード中で使用しても問題になることはほとんどありません。また、数値なので、加算や減算などの演算もできます。ASCIIコードでの各文字の割り当てでは、「'a'」の次は「'b'」になっており、「'z'」まで順番に並んでいます。そのため、「'a'+2」は「'c'」となります。また、大文字のアルファベットと小文字のアルファベットには別々の値が割り当てられているので、大文字と小文字は別々の文字として扱われます。

次のコードは、印字可能な文字をログに出力します。

```
for (int i = 0; i < 128; i++)
{
    // 印字可能な文字か調べる
    if (isprint(i))
    {
        // ログに数値と文字の形で出力する
        NSLog(@"0x%02X (%d) = '%c'", i, i, (char)i);
    }
}
```

関連項目 ▶▶▶

●文字列について ……p.160

変数について

Objective-Cの変数

変数は、値を入れておくことができる場所です。変数の宣言は、次のように、先に「型」を指定し、その後ろに「変数名」を指定します。「=」演算子で初期値を指定することもできます。

```
long l;          // 「long」型の変数「l」を宣言する
int i = 10;      // 「int」型の変数「i」を宣言する。初期値は「10」
```

▶代入

変数への値の代入は「=」演算子を使用します。

```
x = 10;      // 変数「x」に「10」を代入する
x = y;       // 変数「x」に変数「y」の値を代入する
```

▶有効範囲(スコープ)

変数は、宣言された場所によって使用できる範囲が決まっています。使用できる範囲をスコープと呼び、「{」で始まり、「}」で閉じられる範囲をブロックスコープと呼びます。ブロックスコープ内で宣言された変数は、そのブロックスコープ内でのみ使用できます。関数やメソッドも1つのブロックスコープとみなすことができます。また、関数やメソッドの外側で宣言された変数は、宣言されたソースファイル内のどこからも使用できます。

```
int x;
{
    // ここでは変数「y」はまだ宣言されていないため使用できない

    int y;

    // ここでは変数「x」と変数「y」を使用できる
}
// ここでは変数「y」は範囲外のため使用できない。変数「x」は使用できる
```

▶Objective-Cのクラスのインスタンスの変数

Objective-Cのクラスのインスタンスは、次のような構文で記述します。

```
クラス名 *変数名;
```

```
// 「NSString」クラスのインスタンスを入れる変数を定義する
NSString *str;
```

インスタンスの代入は、他の変数と同様に「=」演算子を使用します。ARCを使用しているときは、次の構文で、インスタンスの生存期間を指定するための修飾子を指定できます。

```
クラス名 * 修飾子 変数名;
```

```
// 「NSString」クラスのインスタンスを入れる変数で、弱い参照関係を持つ
NSString * __weak str;
```

指定できる修飾子には、次の種類があります。

修飾子	説明
__strong	強い参照関係を持つ。変数に格納されている間は、解放されない
__weak	弱い参照関係を持つ。変数に格納されていても、インスタンスが解放される場合もあり、解放されると「nil」になる。これを「ゼロ化」と呼ぶ
__unsafe_unretained	弱い参照関係を持つ。「__weak」とは異なり、ゼロ化は行われない
__autoreleasing	返すときに「autorelease」される

COLUMN　ARCを使用しているときの変数への代入

ARC（Automatic Reference Counting）を使用しているときは、変数へのインスタンスの代入は単なるポインタの代入以上の意味を持ちます。ARCを使用しているときに、強い参照関係を持つ変数へのインスタンスの代入は、すでに代入されているインスタンスが解放されて、新しいインスタンスが代入されるという動きになります。そして、他の強い参照関係を持つように指定された変数にそのインスタンスが入っていなければ、そのインスタンスは解放されます。インスタンスが解放されるということは、弱い参照関係を持つように指定された変数に代入されていれば、その変数の値はゼロ化されます。

このように、ARCを使用しているときには、代入は、同時にインスタンスの解放も行われる可能性があるということに注意してください。特に弱い参照関係を持つように指定した変数については注意が必要です。

関連項目 ▶▶▶
● 演算子について …… p.70

演算子について

Objective-Cの演算子

Objective-Cで利用可能な演算子は、C言語と同じです。演算子には優先度がそれぞれ決まっていますが、「(」と「)」を使うことで優先させたい部分を指定できます。筆者は、演算子の処理される順番を明確にしたいため、演算子で定義される優先度に関係なく「(」と「)」を使用して、優先させたい部分を明示するようにしています。

▶算術演算子

算術演算は一般的な記法で記述します。一般的な算術演算と同様に、「乗算」と「除算」は、「加算」と「減算」よりも優先されます。

演算子	説明
*	かけ算
/	割り算
+	加算
−	減算
%	余りを計算

```
i = (10 + 2 * 5) % 3;   // 「2」と計算され、変数「i」に代入される
x = x * y;              // 「x * y」の結果が「x」に代入される
a = b + 10 * c;         // 「10 * c」が先に計算され、
                        // 変数「b」と加算した結果が変数「a」に代入される
z = (x + 10) * 5;       // 「x + 10」が先に計算され、
                        // 「5」と積算した結果が変数「z」に代入される
```

▶インクリメントとデクリメント

インクリメントは変数の値を1増やし、デクリメントは1減らします。

```
int i = 0;

i++;    // 「i = i + 1」と同じ
i--;    // 「i = i - 1」と同じ
```

▶ビット演算子

ビット演算子は、ビット単位での演算を行います。ビット単位の演算は、値を2進数にし、ビット列として計算します。

演算子	説明
|	ORを計算する
&	ANDを計算する
~	NOTを計算する
^	XORを計算する
>>	右ビットシフトを計算する
<<	左ビットシフトを計算する

```
unsigned char x = 5;    // 2進数で「00000101」
unsigned char y = 3;    // 2進数で「00000011」
unsigned char z;
z = x | y;      // 2進数で「00000111」
z = x & y;      // 2進数で「00000001」
z = ~x;         // 2進数で「11111010」
z = x ^ y;      // 2進数で「00000110」
z = x << 5;     // 2進数で「10100000」
z = x >> 1;     // 2進数で「00000010」
```

▶論理演算子と関係演算子

論理演算子は、演算子の左辺値と右辺値について、ANDやOR、NOTを求める演算子です。結果を「偽」(0)か「真」(1)で返します。関係演算子は左辺値と右辺値を比較する演算子です。どちらの演算子も条件分岐などで使用します。

演算子	説明
==	左辺値と右辺値が等しいとき「真」
!=	左辺値と右辺値が等しくないとき「真」
<	左辺値が右辺値未満のとき「真」
>	左辺値が右辺値超過のとき「真」
<=	左辺値が右辺値以下のとき「真」
>=	左辺値が右辺値以上のとき「真」
&&	左辺値と右辺値が両方とも「真」のとき「真」(AND演算)
||	左辺値と右辺値のいずれかが「真」のとき「真」(OR演算)
!	右辺値が「偽」のとき「真」、「真」のとき「偽」(NOT演算)

▶三項演算子

三項演算子は条件式が成り立つとき(「真」となるとき)の値と、成り立たないとき(「偽」となるとき)の値を指定する演算子です。値には数式や関数などを指定することもできます。構文は、次の通りです。

```
(条件式) ? (「真」となるときの値) : (「偽」となるときの値)
```

```
// 変数「y」の値が0よりも大きいとき、関数「doSomething」の戻り値を変数「x」に代入する
// それ以外のときは「0」を代入する
x = (y > 0) ? doSomething(y) : 0;
```

▶省略形

演算子には省略形が使えるときがあります。たとえば、「i = i + 2」は、「i += 2」と書くことができます。このような省略形には次のようなものがあります。

省略形	省略される前の形
a += b	a = a + b
a -= b	a = a - b
a *= b	a = a * b
a /= b	a = a / b
a %= b	a = a % b

省略形	省略される前の形
a &= b	a = a & b
a|= b	a = a|b
a <<= b	a = a << b
a >>=b	a = a >> b

▶「sizeof」演算子

「sizeof」演算子は、型の大きさを返す演算子です。たとえば、「sizeof(short)」の場合、「short」型は16ビットの変数で2バイト必要とするので、「2」が返ります。「sizeof」演算子には、型、もしくは、変数を指定します。構造体を渡した場合には、その構造体の必要サイズが返ります。

▶キャスト

変数やリテラルの型を変換したいときは、キャストを使用します。キャストは、次のような構文で記述します。

```
// リテラルのキャスト
(変換後の型)リテラル

// 変数や定数のキャスト
(変換後の型)変数や定数
```

```
// 浮動小数点数の掛け算の結果を「int」型に変換して代入する
// 小数点以下の値は切り捨てられる
int i = (int)(20.4 * 3.0);
```

キャストで注意しなければいけないことは、変数や定数、リテラルの内容を変換するのではなく、コンパイラが認識する型を変換するだけなので、たとえば、キャストを使って数値型を数字で構成される文字列に変換することはできません。数値から数字で構成される文字列に変換したり、文字列を解析して数値に変換したりするには、専用の関数やメソッドを使用する必要があります。

▶ARCとトールフリーブリッジ

Core Foundationの型とObjective-Cのクラスの中には、キャストするだけで変換することができるトールフリーブリッジに対応した型とクラスがありますが、ARCを使用しているときは単純なキャストではなく、「__bridge」キーワードも使用する必要があります。

```
// 「CFStringRef」型から「NSString」クラスのインスタンスに変換する
CFStringRef cfStr = CFSTR("TEST");
NSString *str = (__bridge NSString *)cfStr;

// 「NSString」クラスのインスタンスから「CFStringRef」型に変換する
NSString *str2 = @"TEST2";
CFStringRef cfStr2 = (__bridge CFStringRef)str2;
```

COLUMN　ARCを使用しているときは「void ＊」へのキャストは行わない

C言語やC++言語で任意のポインタを型やクラスに関係なく格納する変数として「void ＊」型を使用することが多くあり、Objective-Cのクラスのインスタンスも同様に「void ＊」にして格納するということも行われていました。しかし、ARCを使用している

場合は、Objective-Cのクラスのインスタンスは「void ＊」にキャストしたり、「void ＊」型の変数に格納してはいけません。「void ＊」型にしてしまうと、ARCがインスタンスの有効範囲を追うことができなくなってしまいます。

COLUMN　三項演算子と条件分岐

三項演算子は「if」文を使用した条件分岐で置き換えることもできます。たとえば、次のようなコードがあるとします。

```
int a = (x < y) ? calc(x) : calc(y);
```

このコードは、次のように置き換えることもできます。

```
int a;

if (x < y)
    a = calc(x);
else
    a = calc(y);
```

三項演算子で記述するか、「if」文を使用した条件分岐で記述するかは好みの問題もありますが、筆者は次のようなときは三項演算子を使用しています。

- 変数を定義したときに使用する初期値の設定
- 「return」文での関数（メソッド）戻り値
- 関数（メソッド）の引数に記述するとき

ただし、上記のいずれかを満たしていても、条件式や値（もしくは式）が複雑になるときは三項演算子を避け、「if」文による条件分岐にしています。

```
// 変数を定義したときに使用する初期値の設定
int a = (x > 0) ? x : 0;

// 「return」文での関数（メソッド戻り値）
return ((x > 0) ? x : 0);

// 関数（メソッド）の引数に記述するとき
[myObject calc:((x > 0) ? x : 0)];
```

関連項目 ▶▶▶
●変数について …… p.68

定数について

Objective-Cの定数

Objective-Cでは、定数として、列挙、プリプロセッサディレクティブ、「const」修飾子を付けた変数を使用することができます。その他、Objective-Cでは文字列定数を頻繁に使用しますが、これには「NSString」クラスを使用します。プログラム中で定数を宣言せずに、その場で値を記述することもできますが、次のような理由でお勧めできません。

- バラバラに記述すると、変更するときも、すべての場所で記述する必要がある。
- 定数の意味が不明。定数にわかりやすい名前を付けておくことで、後から見てもその値の意味がわかりやすくなる。
- 文字列定数(「NSString」クラス)の場合、毎回、文字列リテラルを記述すると、スペルを間違えてもコンパイルエラーにならないため、間違えたことを見つけるのが大変になる。

なるべく定数は直接、記述せず、定数として宣言し、その宣言した定数を使用することをお勧めします。

▶列挙

列挙の定義には、予約語の「enum」を使用します。

```
// 「enum.h」ファイル
#import <Foundation/Foundation.h>
// 列挙を定義する
enum
{
    // OneObjectTypeの値を2000として定義する
    OneObjectType   = 2000,

    // TwoObjectTypeの値を3000として定義する
    TwoObjectType   = 3000,

    // ThreeObjectTypeの値を3001として定義する
    ThreeObjectType
};
```

「enum」は「{}」で囲んだスコープ内に、「定数名=値」の形式で定義します。複数の項目を列挙するには「,」(カンマ)で区切ります。値を省略した場合は、直前の定数の値に1加算した値が使われます。先頭の定数の値を省略した場合は、先頭の値は「0」になります。定義した値を使用するときは、次のコードのように、値の代わりに定数名を記述します。

```
// 「main.m」
#import <Foundation/Foundation.h>
```

```
#import "enum.h"

int main (int argc, const char * argv[])
{
    @autoreleasepool
    {
        // 列挙を使用するときは数値の代わりに列挙の定数名を書く
        NSLog(@"OneObjectType=%d", OneObjectType);
        NSLog(@"TwoObjectType=%d", TwoObjectType);
        NSLog(@"ThreeObjectType=%d", ThreeObjectType);
    }

    return 0;
}
```

▶プリプロセッサディレクティブ

C言語と同様にプリプロセッサディレクティブの「#define」を使用します。コンパイラは「#define」で定義された定数を値に置き換えてコンパイルします。「#define」では数値だけではなく、文字列なども定義できます。

```
// 「Define.h」ファイル
// 「InitialValue」という名前で「10」を定義する
#define InitialValue    10
// 「InitialString」という名前で「"ABCDEF"」を定義する
#define InitialString   "ABCDEF"
```

「#define」は「定数名　値」の形式で定義します。値と定数名は半角スペースやタブなどで区切ります。1つの「#define」に付き、1つの定数を定義できます。定義した値を使用するときは次のコードのように、値や文字列の代わりに定数名を記述します。

```
// 「main.m」ファイルのコード
#import <Foundation/Foundation.h>
#import "Define.h"

int main (int argc, const char * argv[])
{
    @autoreleasepool
    {
        // 定義した定数を使用するときは、値の代わりに定数名を記述する
        NSLog(@"InitialValue=%d", InitialValue);
        NSLog(@"InitialString=%s", InitialString);
    }
    return 0;
}
```

▶「const」修飾子

変数の型宣言の前に「const」修飾子を付けます。「#define」とは異なり、「変更ができない変数」という扱いなので、「型」の情報を持つことが特徴です。そのため、間違った使い方をしたときに、ある程度、コンパイラがコンパイル時に検出することができます。

```
// 「Const.h」ファイル
// 「InitialValue」という名前で「10」として定義する
const int InitialValue = 10;
```

定義した値を使用するときは次のコードのように、定数名を記述します。

```
// 「main.m」ファイル
#import <Foundation/Foundation.h>
#import "Const.h"

int main (int argc, const char * argv[])
{
    @autoreleasepool
    {
        // 定義した定数を使用するときは、値の代わりに定数名を書く
        NSLog(@"InitialValue=%d", InitialValue);
    }

    return 0;
}
```

▶「NSString」クラス

文字列定数には「NSString」クラスを使用します。ヘッダファイルに直接、定義してしまうと、読み込んだソースファイルごとに別々のインスタンスとなってしまうため、リンクエラーになってしまいます。そのため、ソースファイルに定義し、ヘッダファイルは「extern」修飾子を付けて宣言し、参照するのみにします。

```
// 「StringConstant.m」ファイル
#import "StringConstant.h"

// 「MyIdentifier」という名前の文字列定数を定義する
NSString *MyIdentifier = @"MyIdentifier";
```

```
// 「StringConstant.h」ファイル
#import <Cocoa/Cocoa.h>

// 「MyIdentifier」という名前の定数を参照する
extern NSString *MyIdentifier;
```

```
// 「main.m」ファイル
#import <Foundation/Foundation.h>
#import "StringConstant.h"

int main (int argc, const char * argv[])
{
    @autoreleasepool
    {
        // 定義した定数を使用するときは、文字列の代わりに定数名を書く
        NSLog(@"Constant=%@", MyIdentifier);
    }

    return 0;
}
```


COLUMN 定数の命名規則について

コード中で定数と変数を見分けるために、定数名の名前の付け方を工夫するという方法があります。Mac OS X向けのソフトウェアの開発でよく使われるのは、次の3つです。

- 小文字の「k」で始め、単語区切りに大文字を使用する(「kMyIdentifier」など)
- すべて大文字で記述し、単語区切りに「_」を使用する(「MY_IDENTIFIER」など)
- 先頭にプロジェクト単位のプレフィックスを付けて、単語区切りに大文字を使用する(「NSOrderedSame」など)

このような名前付けのルールを「命名規則」と呼びます。

Foundationフレームワークでは、先頭のプロジェクト単位のプレフィックスを付けて、単語区切りに大文字を使用するの方法が使われています。Carbonフレームワークでは小文字の「k」で始め、単語区切りに大文字を使用する方法が使われています。どの方法を使用してもよいと思いますが、開発プロジェクトチームや開発会社ごとに定義された標準の方法がある場合は、それに従うとよいでしょう。

関連項目 ▶▶▶
- プリプロセッサディレクティブについて …… p.85
- 「Immutable」なクラスと「Mutable」なクラス …… p.133

コメントについて

コメントの記述方法

コメントは、プログラムコード中に記述してもコンパイラには無視される部分です。Objective-Cのコードは、他の言語と比較しても、後から見たときに後から意味をとらえやすいコードです。しかし、それぞれのコードが何をしているのか、何をするためのコードなのか、前提条件などをコメントで残しておくと、後々、保守がやりやすくなります。

Objective-Cでは、コメントの記述方法が2種類あります。

- 「//」と書いた部分から行末までがコメントとなる（本書ではこの方式でコメントを記述している）
- 「/*」から「*/」までがコメントとなる

```
int a = 10;  // これ以降は行末までコメントとして無視される

int b = 20;  /* この部分はコメントとして無視される */

/*
  この行はコメントとして無視される
  このスタイルのコメントは複数行にすることもできる
*/
```

COLUMN　コードをまとめてコメントアウトする

開発を行っていると、一時的にコードをまとめてコメントアウトしたいということがあります。たとえば、デバッグのときに一時的にコードを無効にしたい場合や、仕様変更のときに古い仕様のコードも参考として残したい場合、わかりにくい不具合を修正したときに元のコードを残しておきたい場合などです。このようなときに、4、5行ならば手作業で行ってもよいですが、30行程度になると面倒です。このようなときに筆者がよく利用するのは、Xcodeの「Comment Selection」機能です。この機能は、選択している行をコメントアウトしてくれます。逆に、この機能でコメントアウトしたコードに再度、使用すると、コメントの文字（行頭に挿入された「//」）を削除して元に戻してくれます。「Comment Selection」機能を使用するには、コードを選択して、「Editor」メニューから「Structure」→「Comment Selection」コマンドを選択します。「Command」キー+「/」というショートカットも割り当てられており、ショートカットを覚えておくと便利です。

条件分岐について

Objective-Cでの条件分岐の種類

プログラムコードで非常に重要な構文に、条件分岐があります。Objective-Cでは、予約語の「if」と「switch」を使った条件分岐があります。

▶「if」を使った条件分岐

「if」は、条件式が「真」となるときにのみ実行するという条件分岐です。次のような構文で使用します。

```
if (条件式1)
{
    条件式1が「真」のときに実行されるコード
}
else if (条件式2)
{
    条件式2が「真」のときに実行されるコード
}
else
{
    条件式1と条件式2がどちらも「偽」のときに実行されるコード
}
```

「else if」や「else」は省略することができます。また、「else if」は、複数、使用することもできます。各条件のときに実行されるコードは「{」と「}」で囲んでいますが、これを省略すると、1ステップだけが実行されます。

次のコードは「if」文の条件分岐を使ってカウンタの値が偶数のときは大文字、奇数のときは小文字でアルファベットを出力しています。

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

int main (int argc, const char * argv[])
{
    @autoreleasepool
    {
        // 大文字と小文字を交互に切り替えながらアルファベットを出力する
        int i;
        for (i = 0; i < 26; i++)
        {
            // 偶数のときは大文字で出力する
            if ((i % 2) == 0)
                printf("%c", 'A' + i);
            else
```

```
                printf("%c", 'a' + i);
            }
        }

        return 0;
}
```

このコードを実行すると、次のように表示されます。

```
AbCdEfGhIjKlMnOpQrStUvWxYz
```

▶「switch」を使った条件分岐

定数を列挙した条件分岐のときに使用します。条件式の値による条件分岐を行います。

```
switch (条件式)
{
    case 値1:
        条件式が値1のときに実行されるコード
        break;
    case 値2:
        条件式が値2のときに実行されるコード
        break;
    default:
        条件式の値が「case」で指定したどれにも一致しないときに実行される
        コード
        break;
}
```

各値は「case」で記述します。各「case」は、「break」まで実行されます。そのため、「break」を入れ忘れると、その下の「case」も実行されるので注意してください。これは、長所でもあり、短所でもあります。意図的に「break」を入れない場合もあります。また、「default」の「break」は必須ではありませんが、筆者はわかりやすさから入れています。

次のコードは「switch」を使って、カウンタの値を3で割った余りによって出力する文字列を切り替えています。

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

int main (int argc, const char * argv[])
{
    @autoreleasepool
    {
        int i;
        for (i = 0; i < 5; i++)
        {
            // 3で割った余りによって出力する文字列を変更する
            switch (i % 3)
```

```
        {
            case 0: // 0のとき
                NSLog(@"Zero");
                break;
            case 1: // 1のとき
                NSLog(@"One");
                break;
            case 2: // 2のとき
                NSLog(@"Two");
                break;
        }
    }
}

    return 0;
}
```

このコードを実行すると、次のように表示されます。

```
2012-03-11 22:22:04.255 Switch[4250:403] Zero
2012-03-11 22:22:04.258 Switch[4250:403] One
2012-03-11 22:22:04.259 Switch[4250:403] Two
2012-03-11 22:22:04.259 Switch[4250:403] Zero
2012-03-11 22:22:04.260 Switch[4250:403] One
```

ループ(繰り返し)について

Objective-Cでのループの種類

Objective-Cでは、ループは4種類あります。どれを使用しても、記述方法によって同じ処理を行えますが、それぞれに特徴があり、使い分けた方がわかりやすいコードになります。

▶「for」を使ったループ

「for」を使ったループは、「初期化」「ループする条件式」「各ループごとに実行するコード」の3つを1度に指定できるループです。筆者は、繰り返す回数が決まっているときに使用しています。

```
for (初期化; 条件式; 各ループ後に繰り返すコード)
{
    繰り返すコード
}
```

たとえば、変数「x」をカウンタとして、10回繰り返すときは、次のように記述します。

```
for (x = 0; x < 10; x++)
{
    繰り返すコード
}
```

「for」を使ったループ内で、途中でループを中断したい場合があります。たとえば、実行中のコードでエラーが発生したときなどです。その場合には、「break」を使用します。次の例では、関数「getStatus」の戻り値が負の値のとき、ループを中断するというコードです。

```
for (x = 0; x < 10; x++)
{
    if (getStatus() < 0)
        break;

    ...
}
```

▶「while」を使ったループ

「while」を使ったループは、繰り返しを行う条件式のみを指定します。カウンタの値が不規則に変化する場合や、ループ開始前に、繰り返し回数が不明なときなどに使用します。

```
while (条件式)
{
    繰り返すコード
}
```

たとえば、次のコードでは、関数「getStatus」の戻り値が0以上のときは繰り返すというコードです。

```
while (getStatus() >= 0)
{
    繰り返すコード
}
```

「while」を使ったループも、「for」を使ったループと同様に、「break」で中断することができます。

▶「do」と「while」の組み合わせ

「while」を使ったループの少し変わった形式に、「do」を組み合わせたものがあります。

```
do
{
    繰り返すコード
} while (条件式);
```

「while」だけを使ったループとの違いは、条件式の評価されるタイミングです。「while」だけを使ったループでは、条件式の評価は先頭で行われます。そのため、1度も繰り返すコードを実行せずに、ループを抜ける可能性があります。それに対して、「do」を組み合わせたループでは、条件式の評価は最後に行われます。そのため、繰り返すコードは、最初から条件式が「偽」となるケースであっても、1度は実行されます。どちらがよくてどちらが悪いということはありません。使用する場所で、わかりやすく、理屈に沿っている方を使用すればよいと筆者は思います。

▶「for」と「in」の組み合わせ

Objective-C 2.0から導入されたループです。「高速列挙」と呼ばれます。Objective-Cのコレクションクラスの要素の列挙にのみ使用できます。

```
for (要素を受け取る変数 in コレクションクラスのインスタンス)
{
    繰り返すコード
}
```

詳細は、**《高速列挙を使用して配列・セット・順序付けされたセットからオブジェクトを順番に取得する》**(p.317)を参照してください。

COLUMN 無限ループ

プログラムコード中で無限ループを使用したいケースがあります。無限ループとはいっても、特定の条件が揃えばループを抜けるというものです。抜ける条件がない無限ループではプログラムはフリーズしてしまいます。無限ループも、「for」を使う方法と「while」を使う方法の2種類があります。どちらを使用するべきかは意見が分かれるところですが、筆者は好みで選択すればよいと思います。もし、チームで決められたルールがあれば、それに統一すればよいでしょう。

```
for(;;) // 初期化、条件式、毎回実行するコードのいずれも指定しない
{
    繰り返すコード

    if (条件式)
        break;
}
```

```
while (1) // 常に「真」になるように1を指定する
{
    繰り返すコード

    if (条件式)
        break;
}
```

関連項目 ▶▶▶

- 高速列挙を使用して配列・セット・順序付けされたセットからオブジェクトを順番に取得する… p.317
- 高速列挙を使用して配列や順序付けされたセットからオブジェクトを逆順に取得する…… p.323

プリプロセッサディレクティブについて

Objective-Cでのプリプロセッサディレクティブ

プリプロセッサディレクティブは、コンパイラがソースファイルをコンパイルするときに実行される命令文です。プリプロセッサディレクティブは、プログラムの実行時ではなく、コンパイル時に行われるという点が重要です。これにより、C言語とObjective-Cの両方から参照されるヘッダファイルで、Objective-Cのときだけ定義させるなどの使い方ができます。また、ヘッダファイルの読み込みなどもプリプロセッサディレクティブで行います。

▶ヘッダファイルの読み込み

ヘッダファイルの読み込みには、C言語と同様に「#include」も使用できますが、Objective-Cでは「#import」を使用します。「#include」と「#import」の違いは、「#include」は書かれた回数だけ読み込まれるのに対して、「#import」は1度だけ読み込まれます。そのため、「#import」で読み込む場合には、C言語のヘッダファイルにあるような「インクルードガード」は必要ありません。

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

▶マクロ定義

定数の定義にも使用し、コンパイル時に置き換えるマクロを定義するプリプロセッサディレクティブには、「#define」を使用します。「#define」で定義したマクロを未定義状態に戻すには、「#undef」を使用します。

```
#define MaxFileCount    10  // MaxFileCountを10と定義する
#undef MaxFileCount         // MaxFileCountの定義を削除する
```

▶プリプロセッサディレクティブでの分岐

プリプロセッサディレクティブで定義したマクロを使って、分岐を行うことができます。

```
#if 条件式1
    条件式1が「真」となるときのコード
#elif 条件式2
    条件式2が「真」となるときのコード
#else
    条件式1と条件式2の両方が「偽」となるときのコード
#endif
```

「#elif」と「#else」は、必要がないときは省略することができます。また、「#elif」は、複数、使用することができます。プリプロセッサディレクティブでの条件式には、プログラムコード中の関数や変数は使用できません。条件式もプリプロセッサディレクティブで定義したマクロや定数を使用する必要があります。条件式に「defined」を使用するとマクロが定義されているかどうかを判定します。