C#による Windows システムプログラミング

第2版

北山洋幸◉著

■サンプルファイルのダウンロードについて

本書掲載のサンプルファイルは、一部を除いてインターネット上のダウンロードサービスからダウンロードすることができます。詳しい手順については、本書の巻末にある袋とじの内容をご覧ください。

なお、ダウンロードサービスのご利用にはユーザー登録と袋とじ内に記されている番号が必要です。そのため、本書を中古書店から購入されたり、他者から貸与、譲渡された場合にはサービスをご利用いただけないことがあります。あらかじめご承知おきください。

本書で取り上げられているシステム名／製品名は、一般に開発各社の登録商標／商品名です。本書では、™ および ® マークは明記していません。本書に掲載されている団体／商品に対して、その商標権を侵害する意図は一切ありません。本書で紹介している URL や各サイトの内容は変更される場合があります。

はじめに

本書は、C# の入門レベルを終えた方々やソフトウェア開発に長く従事してきた方々を対象として、C# を使った本格的なシステムを開発するときに悩むようなことがらを中心にまとめた書籍です。

C# は、プログラミング入門者にとって比較的習得が容易な言語ですが、使いこなすとなると話は別です。表面上の簡単な使い方で十分なのか、C# の本格的な使用法までマスターするかで、習得までの労力は大きく異なります。本書は、C# 特有な機能を使いこなすレベルまではいかなくても、いろいろな言語をマスターし、システム開発にも慣れ親しんだ人が、C# を使用してシステムプログラミングを行うのに最低限必要と思われる内容を解説します。たとえば、目的の性能を達成するための方策を知りたい場合や、あるいはオープンソースのプロダクトなどと融合したシステムを開発したい場合などを想定しています。もちろん、C# の初心者に読んでいただいても有益でしょうが、本書は一般の入門書と異なり、ある程度の知識と経験があることを前提として解説しているので、最初に読む本として適しているとは言えません。

本書は、目次からも分かるように、C# や .NET Framework を体系的に学びたい方々を対象とした書籍ではありません。C# の概要を網羅的に学びたい方々には別の書籍にあたることをお勧めします。そのかわり、C# に慣れてしばらくした頃に読み直すと良さそうな題材を厳選しました。一般の C# 入門書では解説しないデータ型の基礎、マネージドコードとアンマネージドコードの融合、並列処理の基礎、並列処理で問題となる GUI とスレッドの関係、データ並列、デリゲートとラムダ式、同期処理、フォーム間やスレッドからのコントロールアクセス、そしてプロセス制御などを解説します。入門書を読破して簡単なプログラムを開発できるようになった人が、実際のシステムを開発するようになったときに読んでほしい内容としました。

本書の対象読者を以降に示します。

- C# でシステムプログラミングを行いたい人
- C# で並列処理や非同期処理について知りたい人
- アンマネージドとマネージドを融合させたい人
- スレッド間などのコントロールアクセスで悩んでいる人
- プロセス間通信を行いたい人

微力ながら、本書が読者のC# プログラミングの理解に役立つことを期待します。

謝辞

出版にあたり、お世話になった株式会社カットシステムの石塚勝敏氏に深く感謝いたします。

2019 年初夏 東大和市桜が丘のカフェにて　北山洋幸

開発現場の実状に即して

何事にも、現実と理想の間には隔たりがあります。その隙間を埋めるのも大事なテーマです。C# と .NET Framework を使用したシステム開発では、アンマネージドコードを知る必要はありません。ところが、現実のシステムはたくさんのソフトウェアの組み合わせによって成り立っています。過去に蓄積した資産を無視して、システムをゼロから開発することは稀です。また、新規開発であっても、システムの一部を外部のライブラリやオープンソースに頼る場合も少なくありません。さらに、すべてをマネージドコードで開発すると、所望の性能を満たせない場合もあります。そのようなケースでは、開発環境を適材適所で使い分けるのも重要です。もちろん .NET Framework では、このような場合を想定して、たくさんのクラスやインターフェースが用意されています。しかし、それでも自身で解決しなければならないことが起きるのが、現実のシステム開発です。このような現実を考え、本書は、あまり触れられることのない、アンマネージドコードとマネージドコードとの融合にもページを割きました。同様に、なるべく C# 内で性能を満足できるように、並列処理の解説にも多くのページを割いています。このような背景から本書は一般の C# や .NET Framework の入門書とは、構成が一味異なっています。

雑感

本書はいくつかの変遷を繰り返しながら完成しました。本書の元となった最初の書籍は、十年以上前に発行されたものです。当時は、C# が一般に使われ始めようとした時期で、C/C++ をメインで使用していた人向けに、C# を使用した場合、どのようにシステムプログラミングを行えばよいか解説しました。おかげさまで増刷を重ね、その後、大幅改訂と増補を行いました。その時期には、相当数の C# 入門書が存在していましたが、それでも毛色が異なった書籍だったのが幸いしたか、それほど部数は多くありませんがコンスタントに売れ続ける書籍となりました。しかし、出版時期の C# や .NET Framework のバージョンとの乖離も目立つようになり、不定期に改訂を行っています。改訂作業は、C# や .NET Framework、そして Visual Studio のバージョンに合わせ、リライトするだけで良いだろうと考えていました。しかし、それが甘い考えだと気づくのに時間は要しませんでした。C# のバージョンアップでは、大幅な変更がなされ従来の記述法が適切な解ではなくなることも少なくなく、全面的な書き直しも発生します。言語や .NET Framework の拡張によって、プログラマの負担が大幅に低減されていることも珍しいことではありません。このため改訂のたびに大きな書き直しが発生します。そんなこんなで、多くの時間を奪われます。さらに、以前の書籍を継承すると大幅にページ数が増大するため、現在では重要でないだろうと思われるトピックを削除する作業にも時間を奪われます。以上のような状況により、非常にバタバタしましたが何とか完成にこぎつけた次第です。

さて、この書籍の元となった書籍を執筆したのは、親族が療養中で頻繁に帰省していた時期で

す。長期の帰省を繰り返していたのが昨日のようです。帰省した当日に、温泉で長湯して高熱を出してしまったのも想い出となりました。私の田舎は温泉が出やすいため、町で運営している温泉を銭湯のように使用します。熱を出した日は長旅に加え湯冷めしないことを過信し、つい油断をしてしまいました。帰省したときに、お風呂を使うことはなく毎日温泉へ出かけます。ゆったりお湯につかったあと、くつろぎの空間でテレビを眺めながら、飲み物をゆったり飲む時間は東京では味わえません。テレビから流れる番組も、ラジオから流れる音楽も東京と変わらないのに、時間はゆったり流れ東京とは別物のように感じます。

田舎の風景は相変わらず昔の面影のままですが、空き家が増え道路だけが立派になっていきます。道路は立派になりますが、それを活用する公共サービスが充実されているかは良く分かりません。完全に車社会化が進んでおり、鉄道も学生など限られた人たちが利用するだけのようです。そういえば、帰省した時に、最寄りの駅を利用したことはなく、空港からはバスかレンタカーを使っています。このような状況が車を持たない高齢者にとって本当に良いのか疑問に思うときもあります。都会に住んでいる人はバスやタクシーを利用したら良いだろうと考える人も多いでしょうが、タクシーさえ呼べない地域も日本には少なくないでしょう。私の故郷もタクシーを気軽に呼べる環境ではありません。ただ、そこは田舎の助け合いで、うまく機能しているのでしょう。

時が経つに連れ、高齢化が進み、若い人は少なくなっています。小学校も中学校も閉鎖が続き、高校の入学者も減少の一途です。まるで日本の少子化を先取りした様子を見ているような気分にもなります。村の空き家も増え、幼少のころの賑わいは遠い昔になりました。昔は子供たちが走り回っていた海岸、海、そして広場に出かけても、そこには静寂だけがあり、小一時間滞在しても誰一人も現れません。私の家も空き家になって久しく、すでに玄関への通路も竹林になり、家に入ることさえ難しくなりました。家の中は朽ち果てていると思いますが、大きな台風に幾度となくさらされても未だに建っています。もう少し若ければ、U ターンも考えましたが、実家が空き家になるのが早すぎました。

田舎に帰るたびに、私の心はいつもリフレッシュされます。東京で悩んでいたことが、なぜ悩みなのか分からなくなります。単純に現実を逃避できたためなのか、それとも田舎には何か魅力があるのか、まだ自分で把握できていません。言えることは田舎に帰ると、時間の流れがゆっくりとなることです。もっとも田舎に帰っている期間は自身が休暇なのであって、日常を過ごしている人達とは、また違った感想があるのでしょう。田舎は東京より、生活コストがはるかに安く、リタイヤ後の環境としては良さそうです。全国の調査でも住みやすさランキングの上位にランクされますので U ターンも良いのですが、いろいろなしがらみがあって実現しそうもないです。変なストレスもなく、精神的にも肉体的にも健康を維持できそうな気がします。もう少し若ければ職場を田舎に移したのですが、もうその時期は過ぎました。

数年前の帰省時のことです。東京に戻るときに、母に「明日、帰るよ」と言ったら「もう、帰

るのは東京か！？」と問われました。一瞬、意味が分からなかったのですが、「ああ、帰るという表現がおかしかったんだな」と分かるまで数秒とかかりませんでした。確かに、何気なく“帰る”という表現を使っていますが、私の“帰る”場所は、東京なのか田舎なのか自分でもわかりません。生活した年数は、遥かに東京が長くなりました。しかし、自分の場所は実家にあるような気がするのは何故なんでしょう。東京から田舎に行くときも、「田舎に帰る」と表現するし、田舎から東京に行くときも「東京に帰る」と表現していることに、母の一言で気がつきました。私の帰る場所は何処なのか未だ不明です。こんなたわいのないことを飛行機の中で考えながら東京へ戻ったのも遠い想い出となりました。

室生犀星の「ふるさとは遠きにありて思ふもの」の解釈はいろいろあるようですが、自分なりの解釈ができてきた気がします。同じ内容なのに、20代で感じたときと現在ではずいぶん違ったものとなりました。

本書の使用にあたって

開発環境、および、実行環境の説明を行います。

Windows バージョン

Windows 10 を使用します。Windows 8.1 などでも問題ないでしょうが確認は行っていません。

Visual Studio バージョン

無償の Visual Studio 2019 Community を使用します。他のバージョンでも問題ないでしょうが、.NET Framework との関係もあり、最新の環境を使用することを勧めます。

用語

クラスとオブジェクト

クラスとオブジェクトはなるべく使い分けています。クラス、オブジェクト両方に適用できる内容については、クラス、オブジェクトを省いている場合もあります。

クラスとインスタンス

クラスとインスタンスもなるべく使い分けています。クラス、インスタンス両方に適用できる内容については、クラス、インスタンスを省いている場合もあります。

オブジェクトとインスタンス

オブジェクトとインスタンスもなるべく使い分けていますが、インスタンスよりオブジェクトが適当と思われる部分ではオブジェクトを使用しています。

フォームとウィンドウ

フォームとウィンドウは同じものを指します。主にデザイン時にはフォーム、実行時にはウィンドウと記述しています。これは、C# 特有の説明ではフォームを使用し、Windows 全体にかかわる実行などの表現はウィンドウが適切と判断したためです。

メインスレッドとワーカースレッド

本書ではユーザーインターフェースを受け持つスレッド（UI スレッド）をメインスレッド、そうでないスレッドをワーカースレッドと表示しています。基本的に UI スレッドがメインスレッドになるとは限りませんが、便宜上、このような表記を用いました。

タスクとスレッド

スレッドとタスクを共存して使用しています。本来はスレッドであっても .NET Framework のクラス名がタスク（Task）のため、タスクと表現した方が適切と思われる部分ではタスクを使用します。

フィールドとメンバ

C# では、C++ のメンバに相当するものをフィールドと呼びます。本書では、メンバという記述を主に使用し、C# に特有な部分の解説でフィールドと記述しています。

デリゲート

クラスなどと異なり、デリゲートには宣言と実体の区別がありません。両方ともデリゲートと表現しているので、宣言と実体の違いは文脈から判断してください。

マネージとマネージド

英文では managed ですが、和文ではマネージが採用されています。本書では英文に倣って、マネージド、アンマネージドと記述します。

ユーザーアカウント

最近の Windows ではユーザーやアカウントの管理が強化されています。たとえば、「標準ユーザー」ではプログラムのインストールやアンインストールは制限されます。コンパイラや OpenCV のインストールやセットアップで警告が出ることがあるので、なるべく「管理者」で実行してください。もちろん、管理者アカウントを使用する場合、危険なこともできるので十分注意してください。

Any CPU か x64、あるいは x86

本書では x64 を中心にチェックしました。これはオペレーティングシステムのビット数が 64 ビットだったためです。もちろん、自身で管理すれば x86 でも問題ありません。DLL などの例もあるので、組み合わせを間違えないようにしてください。

URL

URL の記載がありますが、執筆時点のものであり、変更される可能性もあります。リンク先が存在しない場合、キーワードなどから自分で検索してください。

参考文献 / 参考サイト

1. MSDN Web サイト「.NET 開発」（http://www.microsoft.com/japan/msdn/）
2. Microsoft デベロッパーセンター（https://msdn.microsoft.com/developer-centers-msdn）
3. 「.NET フレームワークのための C# システムプログラミング」、カットシステム、北山洋幸著
4. 「Parallel プログラミング in .NET Framework 4.0」、カットシステム、北山洋幸著
5. 「Win64 API システムプログラミング」、カットシステム、北山洋幸著

目 次

はじめに iii

第 1 章● C# 概論……1

1-1 C# の基本 1
1-2 データ型 2
●値型と参照型……3 ●値型の一覧……3 ●参照型の一覧……4
1-3 値型とは 4
1-4 参照型とは 5
1-5 C# の型と Alias の関係 6
1-6 C# の型 8
●bool 型……8 ●byte 型……9 ●sbyte 型……9 ●char 型……10
●decimal 型……10 ●double 型……10 ●float 型……11 ●int 型……11
●uint 型……12 ●long 型……12 ●ulong 型……12 ●object 型……12
●short 型……13 ●ushort 型……13 ●string 型……13 ●var 型……13
1-7 文字列型 15
1-8 配列 19
1-9 構造体 27
1-10 引数 30
●基本動作……30 ●オブジェクトを渡す……36 ●配列を渡す……40
1-11 デリゲート入門 46
●単純な例……47 ●定義済みデリゲート……48 ●デリゲートを引数で渡す……49
●コマンド解析の例……51

第 2 章●マネージドとアンマネージド……55

2-1 アンマネージド呼び出しの概要 55
●関数名のエクスポート……56

2-2 マネージドプログラムとアンマネージドプログラムの連携 57
●DLL の開発……58 ●呼び出し側の開発……66 ●実行……71
●64 ビット対応……74 ●ソリューションで管理……78 ●XAML 編集……85
2-3 DLL のメソッド名を変更する 87
2-4 .DEF ファイルを使ったエクスポート 89
●DLL の開発……89 ●呼び出し側の開発……93
●.DEF ファイルを使う場合の利点と欠点……94
2-5 アンマネージドからマネージドを利用 94
●DLL の開発……95 ●呼び出し側の開発……96
2-6 マネージド／アンマネージド間のデータ交換 98
●C# とアンマネージド間のデータ型対応……98
●マネージドとアンマネージドのデータサイズ……99 ●データモデル……100
●データサイズの比較……100
2-7 マネージドからアンマネージドへデータを渡す 103
●Byte 型と Char 型……103 ●いろいろな整数型……105 ●浮動小数点型……108
2-8 アンマネージドからマネージドへデータを返す 110
●byte 型と int 型……110
2-9 文字列の受け渡し 113
2-10 構造体の受け渡し 115
2-11 配列の受け渡し 117

第 3 章●並列処理……121

3-1 並列処理の概要 121
●並列化する目的……121 ●オーバーヘッド……122
●データアクセス競合……122 ●スケーラビリティ……122 ●スレッド……123
●スレッドの応用……125
3-2 シンプルスレッド 125
●実行……128
3-3 Task で記述 128
●タスク並列ライブラリ概論……130
3-4 暗黙的な起動 130
3-5 ラムダ式で記述 131
3-6 スレッドに値を渡す 132

●実行……134

3-7　スレッドに値を渡す（Task で記述）……135

●ラムダ式に引数を渡す……137

3-8　スレッドから情報を受け取る……139

●Thread クラスで記述……140　●実行……143

3-9　タスク配列……143

●単純なタスク配列……143　●タスク配列と返却値参照……145
●全タスクの処理が同じ……146

3-10　タスク継続……148

●単純なタスク継続……148　●複数のタスクの継続……150

3-11　入れ子タスクと子タスク……152

●入れ子タスク……153　●子タスク……154

第 4 章●デリゲートとラムダ式……157

4-1　デリゲート……157

●デリゲートとは……157　●定義済みデリゲート……160
●デリゲートの実用例……162　●インスタンスのメソッド……164
●マルチキャストデリゲート……166

4-2　ラムダ式……168

●通常のプログラム……168　●デリゲートで記述……169　●匿名メソッド……170
●ラムダ式（1）……170　●ラムダ式（2）……171

第 5 章●データ並列化……173

5-1　データ並列の基礎……173

●単純な Parallel.For ループ……174　●単純な Parallel.ForEach ループ……176

5-2　ループからの脱出……177

●Stop を使用して Parallel.For ループから抜ける……178
●IsStopped プロパティを調べる……179
●Break を使用して Parallel.For から抜ける……180
●LowestBreakIteration プロパティを調べる……182

5-3　スレッドローカル変数……186

●スレッドローカル変数を使用した Parallel.For ループ……186

5-4　ループ取り消し……190
●Parallel.For ループを取り消す……190

第 6 章●並列処理と GUI……193

6-1　並列処理と GUI 更新の概要……193
6-2　シングルスレッド……195
6-3　スレッドとタイマーで監視……199
●実行……203　●WPF で開発……203　●実行……207
6-4　BackgroundWorker を使う方法……207
●実行……210　●WPF で開発……211　●実行……212
6-5　Thread クラスと Invoke……213
●実行……216　●Invoke の判断をメソッドへ……217　●WPF で開発……219
6-6　Task クラスと Invoke……221
●WPF で開発……223
6-7　非同期メソッド……225
●async 修飾子……225　●await 演算子……226　●実行……228
6-8　非同期メソッド応用……229
●並列処理しない場合……230
●async 修飾子と await 演算子を使用したプログラム……234　●実行……237
●非同期プログラムの実行……239
6-9　Task クラスと Invoke（GUI を頻繁に更新）……240
●実行……242　●WPF で開発……243
6-10　非同期メソッド（GUI を頻繁に更新）……244
●実行……245
6-11　非同期メソッド応用（GUI を頻繁に更新）……246
●実行……247
6-12　スレッド間でメッセージ通信（1）……249
6-13　スレッド間でメッセージ通信（2）……253
●実行……260
6-14　スレッド間でメッセージ通信（3）—Task クラス……262
6-15　BackgroundWorker を使ったスレッド間のコントロールアクセス……264
●実行……269

6-16 スレッドと Queue クラスによるスレッド間のコントロールアクセス....270
●実行……278
6-17 バウンズ（BackgroundWorker）....279
6-18 バウンズ（Invoke）....285
6-19 CreateGraphics を使ったコントロールアクセス....288

第 7 章●同期……291

7-1 Interlocked クラスで同期....291
●変数へのインターロックドアクセス……297 ●実行……298
●Task クラスで記述……299
7-2 終了監視....301
●実行……303 ●Task クラスで記述……303
7-3 Monitor クラスによる同期....304
●クリティカルセクションの基礎……305 ●実行……309
7-4 lock 文による同期....310
●プログラムの概要……312 ●実行……315
7-5 非同期呼び出し....317
●実行……320
7-6 同期呼び出し....321
●実行……324
7-7 BeginInvoke メソッド....325
7-8 EventHandler デリゲートによる同期呼び出し....327
●実行……329
7-9 ManualResetEvent と同期....329
●実行……333
7-10 AutoResetEvent と同期....333
7-11 スレッドプール....335
●実行……339
7-12 コールバックでメソッド完了を知る....340
●実行……343
7-13 コールバックとたくさんの非同期呼び出し....345
●実行……347

7-14 volatile キーワード 348

7-15 ReaderWriterLock によるクラス同期 350

●実行……356

7-16 ミューテックス 356

●実行……359

7-17 ミューテックス（GUI 版） 361

●実行……365

7-18 セマフォ 366

7-19 コンカレントコレクション 368

●ConcurrentQueue クラス……368　●逐次プログラム……368

●ConcurrentQueue クラスを使う……369　●タスク配列と返却値……371

●ConcurrentStack……373　●ConcurrentBag……374

●コンカレントコレクション応用例……375

第 8 章●フォーム間のコントロールアクセス……385

8-1 親フォームから子フォームのラベル表示 385

8-2 親フォームから子フォームのコンストラクタでラベル表示 389

8-3 子フォームから返却値を受け取る 391

●実行……393

8-4 親フォームから子フォームのコントロールをアクセス 394

●実行……397

8-5 子フォームから親フォームを直接呼び出す 398

●実行……401

8-6 子フォームから親に Paint イベントを送る（1） 402

●実行……405

8-7 子フォームから親に Paint イベントを送る（2） 405

●実行……408

8-8 イベントハンドラで子フォームから親に通知 409

●実行……412

8-9 フォーム間のメッセージ通信によるコントロールアクセス 413

●実行……418

第 9 章●プロセス……419

9-1　プロセスの起動 419
●実行……422
9-2　プロセスを起動し同期で完了を待つ 424
●実行……426
9-3　プロセスを起動し非同期で完了を待つ 428
●実行……432
9-4　ミューテックスでインスタンスの多重起動を禁止 433
●実行……436
9-5　セマフォでプログラムの多重起動を禁止 437
●実行……439
9-6　プロセス名で多重起動を禁止 439
●実行……441
9-7　DLL によるプロセス制御 441
●DLL の開発……442　●呼び出し側の開発（1）……444　●実行……446
●呼び出し側の開発（2）……447　●実行……448
9-8　DLL によるプロセス間通信 450
●DLL の開発……451　●送信プログラム……453　●受信プログラム……456
●実行……460
9-9　標準 I/O を使用したプロセス間通信 463
●サーバプログラム……464　●クライアントプログラム……467　●実行……469
9-10　ネットワークを使用したプロセス間通信 470
●サーバプログラム……471　●クライアントプログラム……474　●実行……476

索 引 481

1

C# 概論

1-1 C# の基本

C# の解説に入る前に、.NET Framework について簡単におさらいします。「.NET Framework とは何か」と問われれば、まずは、クラスライブラリの集合体であるということができます。クラスライブラリを使用すれば、システムのさまざまな機能にアクセスできます。.NET Framework が提供するクラスライブラリは、完全にオブジェクト指向となっています。さらに、.NET Framework はプログラムの実行環境であるということもできます。.NET Framework で開発したプログラムは .NET Framework コードに変換され、そのコードを .NET Framework 環境が実行します。

このクラスライブラリの集合体と実行環境を合わせたものを、.NET Framework と呼んで差し支えないでしょう。実行環境は Java に似ていますが、Java が Java で閉じているのに対して、.NET Framework 環境は、Visual Basic であろうが、C++ であろうが、C# であろうが、.NET Framework で開発されている限り言語に依存しない点が異なります。言語が混在するアプリケーションでも、デバッグはシームレスに行われます。図 1.1 に、Windows 上の .NET Framework の概要を示します。

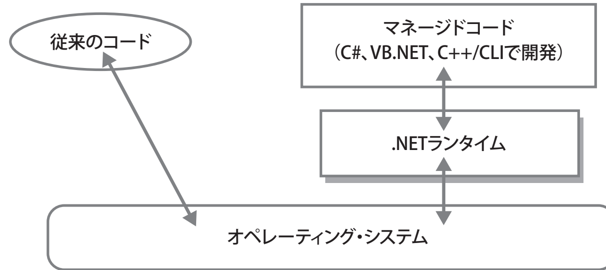

図1.1●.NET Frameworkの概要

C# は、Microsoft 社によって .NET Framework と同時に開発された新しい言語です。それだけに、.NET Framework との親和性もひときわ高いといえるでしょう。

本章では、まず C# のデータ型の概要と、参照型、値型の説明を行い、それから文字列の扱い、配列、構造体、メソッド呼び出しなどの引数の扱い、およびデリゲートなどの基本を解説します。

1-2 データ型

他の言語と同様に、C# にもデータ型があります。int などの型名は、予約語として C# 言語に予約されています。

整数型には、ビット長、符号の有無でいくつかの型が存在します。また、IEEE754 準拠の浮動小数点数が用意されています。他の C 系言語と異なる型として、金額などの財務計算に適した decimal 型があります。この型は小数も扱うことができます。2 進数の浮動小数点数を使用すると 10 進数の小数以下に丸めなどの誤差の生じる恐れがありますが、decimal 型を使用すればそのような誤差は発生しません。

文字型の char もあります。ただし C# の char 型は、C/C++ などの char とは異なり、常に 2 バイトです。内部コードには Unicode が使用されます。C/C++ では、char をバッファなどに使用して内部を 1 バイトずつスキャンする場合もありますが、C# の char 型はあくまでも文字を扱うデータ型という位置付けになっています。C# でバッファを単純にバイト単位で処理するには、byte 型もしくは sbyte 型を使用します。

C# には文字列型も用意されています。C/C++ では char 型の配列で文字列を保持しますが、C# には専用の string 型が用意されています。string 型は、これまでの型と違い、参照型です。「値型」と「参照型」の違いについては後述します。他にもさまざまな型がありますが、個別の型の詳細についても後述します。

各データ型は、C# ではサイズが固定されています。たとえば int 型の場合、C ではプラットフォームや処理系によって異なりますが、C# では常に 4 バイトを占めます。

変数の宣言方法やスコープなどは、C/C++ とほぼ同じです。しいて異なる点をあげれば、クラス内で宣言する変数を、一般的にはメンバ変数と呼びますが、C# ではフィールドと呼びます。メソッドやローカル変数などは一般的な呼び方と同じなので、本書の記述ではメンバ変数も使用します。

値型と参照型

C# のデータ型は、値型と参照型の 2 つに大きく分類できます。たとえば int 型は値型、string 型は参照型です。この 2 種類の型は、メモリを確保する方法で大きく異なります。C# によるプログラミングでは、使用する変数がどちらの型であるかを意識しなければなりません。C# のデータ型を値型と参照型に分けて以降に示します。

値型の一覧

表1.1●整数型

型	範囲	サイズ
sbyte	−128 ～ 127	符号付き 8 ビット整数
byte	0 ～ 255	符号なし 8 ビット整数
char	U+0000 ～ U+ffff	Unicode16 ビット文字
short	−32,768 ～ 32,767	符号付き 16 ビット整数
ushort	0 ～ 65,535	符号なし 16 ビット整数
int	−2,147,483,648 ～ 2,147,483,647	符号付き 32 ビット整数
uint	0 ～ 4,294,967,295	符号なし 32 ビット整数
long	−9,223,372,036,854,775,808 ～ 9,223,372,036,854,775,807	符号付き 64 ビット整数
ulong	0 ～ 18,446,744,073,709,551,615	符号なし 64 ビット整数

表1.2●浮動小数点型

型	範囲	サイズ
float	$\pm 1.5 \times 10^{-45}$ ～ $\pm 3.4 \times 10^{38}$	7 桁
double	$\pm 5.0 \times 10^{-324}$ ～ $\pm 1.7 \times 10^{308}$	15 ～ 16 桁

表1.3●decimal型

型	おおよその範囲	有効桁数
decimal	$\pm 1.0 \times 10^{-28}$ ～ $\pm 7.9 \times 10^{28}$	28 ～ 29 桁

表1.4●bool型

型	おおよその範囲	有効桁数
bool	true または false	—

参照型の一覧

表1.5●参照型

型	範囲	サイズ
object	—	—
string	—	—

1-3 値型とは

値型とは、変数の宣言を行った時点で、そのデータ型の領域自体が確保される型を指します。リスト 1.1 のプログラムを例に説明します。

リスト1.1●値型の宣言

```
using System;

public class Class1
{
    public static void Main()
    {
        short s=10;
        long L=10L;

        Console.WriteLine("s={0}",s);
        Console.WriteLine("L={0}",L);
    }
}
```

このプログラムでは、s の宣言部分で、実際の short 型変数 s の領域 2 バイトが確保され、L の宣言部分で、実際の long 型変数 L の領域 8 バイトが確保されます。

値型については、ごく一般的な変数の宣言であり、特に変わったことはありません。しかし、次節で説明する参照型は、一般的な言語とは多少異なる点があります。

1-4 参照型とは

参照型は値型と違い、変数の宣言部分で、そのデータ型の領域が確保されるわけではありません。C# で予約されている参照型のデータ型は、string 型と object 型の 2 つのみです。しかし、object 型を基底としたクラスなどが多数存在するため、実際のプログラミングで参照型を扱う機会は決して少なくありません。

C# では、基本的にポインタを使用することはできません。「基本的に」と書いたのは、互換性確保のためか、限定的な方法でポインタの使用が許されているからです。しかし、C# では、十分な理由がない限りポインタを使用すべきではないでしょう。

では、C# ではクラスのオブジェクト（インスタンス）にどのようにアクセスするのでしょうか。

リスト1.2●疑似コード

```
MyClass Class1;

Class1 = new MyClass();
```

C# で参照型のオブジェクトを作成するには、new を使用します。リスト 1.2 の例で示すと、Class1 の宣言部分では、MyClass のオブジェクトを参照するメモリが確保されるだけで、オブジェクトの実体は作られません。

object 型は C# の標準型、ユーザー定義型の基底となる親の型です。クラスについては、本書を読み進めるうちに理解が進むでしょうから、ここで細かい説明は行いません。

もう 1 つの string 型ですが、これは文字列の処理に使用します。C/C++ 言語の学習では、第 1 の壁がポインタ、第 2 の壁が文字列処理ともいわれますが、C# ではごく自然に文字列を処理できます。リスト 1.3 を見てください。

リスト1.3●C#による文字列処理

```
using System;

public class Class1
{
    public static void Main()
    {
        string s;

        s="test";
        Console.WriteLine("s={0}",s);
```

```
            s="Spacesoft";
            Console.WriteLine("s={0}",s);
        }
    }
```

sの宣言部分では、文字列への参照情報を保持するための領域が割り当てられます。そして、「=」によって文字列がsに代入され、WriteLineメソッドで出力されます。このプログラム実行結果は次のようになります。

```
s=test
s=Spacesoft
```

注意しなければならないのは、sへの文字列の代入は、sに割り当てられた領域に文字列の実体を格納することではないという点です。正確には、その文字列への参照情報（ポインタ）が格納されることになります。この例では、宣言によってsの領域が割り当てられた後、まず文字列"test"の実体への参照情報がsに格納され、次に文字列"Spacesoft"の実体への参照情報に更新されます。

このように、値型と参照型では変数に格納するデータの取り扱いに大きな相違点があります。

1-5 C#の型とAliasの関係

C#の組み込み型のキーワードを表1.6に示します。これらはSystem名前空間に組み込まれた型のAlias（エイリアス、別名）です。

表1.6●C#の型と.NET Framework型の対応

C#型	.NET Framework型
bool	System.Boolean
byte	System.Byte
sbyte	System.SByte
char	System.Char
decimal	System.Decimal
double	System.Double
float	System.Single
int	System.Int32

C# 型	.NET Framework 型
uint	System.UInt32
long	System.Int64
ulong	System.UInt64
object	System.Object
short	System.Int16
ushort	System.UInt16
string	System.String

表1.7●暗黙型

型	範囲	サイズ
var	—	—

object 型と string 型以外はすべて単純型です。また、C# 型と Alias は相互に交換可能です。たとえば、整数の short と System.Int16 を使用して宣言した変数は同じ性質を持ちます。C# 型の実際の型を表示するには、GetType メソッドを使用します。プログラム例をリスト 1.4 に示します。

リスト1.4●ch01¥01type¥00GetType¥ConsoleApp¥Program.cs（一部）

```
            int i = 0;
            System.Int32 i32 = 0;

            Console.WriteLine("i = " + i.GetType());
            Console.WriteLine("i32 = " + i32.GetType());
```

実行結果は次のようになります。int 型も System.Int32 型も System.Int32 型であることが分かります。

```
i = System.Int32
i32 = System.Int32
```

1-6 C# の型

C# で使用されるデータ型の詳細を説明します。

bool 型

bool 型は System.Boolean のエイリアスです。bool 型変数には bool 値（true または false）を代入できます。

リスト1.5●ch01¥01type¥01Bool01¥ConsoleApp¥Program.cs

```
using System;

namespace Bool01
{
    class Program
    {
        static void Main(string[] args)
        {
            bool bRval = true;

            if (bRval)
                Console.WriteLine("bValはtrueです。");
            else
                Console.WriteLine("bValはfalseです。");
        }
    }
}
```

C++ などではリスト 1.6 のような記述が可能ですが、C# では無効となります。

リスト1.6●ch01¥01type¥02Bool02¥ConsoleApp¥Program.cs（一部）

```
            int iRval = 1;

            if (iRval)  // ←コンパイルエラーになる
                ⋮
```

byte 型

byte 型は、0 から 255 までの整数を格納できる符号なし 8 ビット整数型です。byte から short、ushort、int、uint、long、ulong、float、double、decimal へは、暗黙の型変換が組み込まれています。逆に、byte に入る値より大きな数値型から byte への暗黙の型変換はできません。必要な場合は明示的にキャストしてください（以下のサンプルコードは ch01¥01type¥03Byte01¥ConsoleApp¥Program.cs 参照）。

```
byte b1;
b1 = 10;
```

は問題ありませんが、

```
byte b1;
b1 = 300;
```

はコンパイルエラーとなります。代入演算子の右側にある算術式が int 型として評価されるためです。このエラーを抑止するには、キャストと unchecked 文を使用します。

```
byte b1;
unchecked
{
    b1 = (byte)300;
}
```

次の例でも、b1*b2 は int 型と解釈されるため、コンパイルエラーとなります。

```
byte b1 = 10, b2 = 20;
byte b3 = b1 * b2;          //←コンパイルエラー
```

このエラーを解決するには、キャストを使用します。

```
byte b1 = 10, b2 = 20;
byte b3 = (byte)(b1 * b2);      //←明示的にキャストすればOK
```

このように、C# の型チェックは厳しく、型変換がらみでコンパイルエラーとなることが少なくありません。基本的に、型変換が起きる場合は明示的にキャストするとよいでしょう。特に、オーバーフローが起きるような、大きな型から小さな型への代入などは注意が必要です。

sbyte 型

sbyte 型は、−128 から 127 までの符号付き 8 ビット整数を格納します。キャストやコンパイルエラーなどについては、符号付きの byte と考えればよいので省略します。

char 型

C# の char 型は 16 ビットで Unicode 文字を格納できます。char 型の定数は、文字リテラル、16 進数（エスケープシーケンス）、Unicode 表現を記述できます。あるいは、数値をそのまま文字コードへキャストできます。

リスト 1.7 は、さまざまな方法で char 型変数に文字 A を代入するコードです。C/C++ の char と混同しないように気を付けてください。

リスト1.7●文字'A'を代入する

```
char cLiteral = 'A';            // リテラル
char cHex     = '¥x0041';       // 16進数
char cCast    = (char)65;       // キャスト
char cUnicode = '¥u0041';       // Unicode
```

char 型変数は、ushort、int、uint、long、ulong、float、double、decimal へ暗黙の型変換ができます。

decimal 型

decimal 型は、10 進数計算のために用意された 128 ビットのデータ型です。財務や金融の計算で、浮動小数点数では無視できない誤差が発生するような場合の使用に適しています。メインフレームなどでは CPU が 10 進数命令（BCD パックド演算命令など）を装備していますが、一般的なパーソナルコンピュータには 10 進数命令がないため、言語でサポートしています。本書の主眼であるシステムプログラミングにはあまり関係がないため、詳細は省きます。

double 型

double 型は、64 ビットの浮動小数点数を格納します。代入演算子の右側にある実数値リテラルは、double 値として扱われます。整数を double 型として扱いたい場合は、サフィックスに d または D を使用します。

リスト1.8●ch01¥01type¥04Double01¥ConsoleApp¥Program.cs（一部）

```
            double d = 3d;
```

1 つの式に整数型と浮動小数点型の数値を混在させることができます。整数型と浮動小数点型が混在する場合、整数型は浮動小数点型に変換されます。