

C# Basic.

A beginner guide for C#

Mine Archive

Outline

- 1. C 言語と C# の違い 3
- 2. オブジェクト指向プログラミングとは 9
- 3. 文字列とコレクションとジェネリック型 18
- 4. C 言語とは異なるコードスタイル 24

C 言語と C# の違い

型システムの違い

C の型	C# の型	値の制限
int	int	-2147483648 ~ 2147483647
float	float	$\pm 1.5e-45 \sim \pm 3.4e+38$
double	double	$\pm 5.0e-324 \sim \pm 1.7e+308$
long	long	-9,223,372,036,854,775,808 ~ 9,223,372,036,854,775,807
char	char	0 ~ 65,535 (ascii も行ける)
該当なし	bool	true or false
char*	string	文字列

加えて、unsigned 型も存在する。

メモリ管理

- C 言語では malloc, realloc を使ってメモリを手動で確保していた。
 - ▶ メモリ管理の責任がプログラマにある。
 - ▶ メモリリークが発生する可能性がある。
 - ▶ 不適切な free()呼び出しでダブルフリーエラーが発生することもある。
- C# ではガベージコレクション(GC)によって自動管理される。
 - ▶ 今後使われないと考えられるメモリは自動的に free()される感じ。
 - ▶ プログラマはメモリ管理の細かい部分を気にする必要がない。
 - ▶ メモリリークのリスクが大幅に低減される。
- 新しく配列を取るときにいちいち長さとか考える必要がない。
 - ▶ 配列のサイズを動的に確保できる。
 - ▶ List などのコレクション型を使えば、サイズ変更も容易。

名前空間と using ディレクティブ

名前空間(Namespace)は、コードを論理的に整理・管理するための仕組み。同じ名前の型が複数存在する場合の名前衝突を避けることができる。

using ディレクティブを使うことで、名前空間内の型を直接参照できる。

C 言語	C#
<code>#include <stdio.h></code>	<code>using System;</code>
ヘッダファイルをインクルード	名前空間をインポート
標準ライブラリの宣言を取り込む	名前空間内の型を直接参照可能にする
stdio.h に書かれてる関数を使えるようにする	System.*に含まれているものを System を書かずに使えるようにする

名前空間と using ディレクティブ (ii)

例:

```
using System;

namespace MyApplication
{
    public class Program
    {
        public void Print()
        {
            //System.Console.WriteLine("Hello!");
            Console.WriteLine("Hello!");
        }
    }
}
```

名前空間と using ディレクティブ (iii)

```
public class Main
{
    public static void Main()
    {
        // 上でMyApplicationの中にProgramを定義しているから
        // MyApplication.Programとして使用できる。
        var program = new MyApplication.Program();
        program.Print();
    }
}
```

- using System; により、System 名前空間の型(Console など)を直接使用可能
- namespace MyApplication { } で、自分のコードを名前空間内に配置

オブジェクト指向プログラミングとは

クラス

- C 言語の構造体関数を持つようになった感じ
- クラスはデータ(フィールド)とそれを操作する処理(メソッド)を 1 つの単位にまとめたもの
- オブジェクト指向プログラミングの中心的概念
- アクセス修飾子(public, private など)でカプセル化が可能
 - ▶ public: 外部からアクセス可能
 - ▶ private: クラス内からのみアクセス可能
- コンストラクタによってオブジェクトの初期化が可能
- インスタンスを生成することで、クラスの具体的な実体が作られる

継承

- 既存のクラス(基底クラス)の機能を引き継ぎ、拡張したクラス(派生クラス)を作る仕組み
 - ▶ コードの再利用性を高める
 - ▶ 共通のインターフェイスで多態性(ポリモーフィズム)を実現できる
 - ▶ C# では「単一継承」(クラスは1つのクラスだけを継承可能)。
- 用語
 - ▶ 基底クラス(Base class): 継承される側
 - ▶ 派生クラス(Derived class): 継承する側
 - ▶ protected: 同一クラスおよび派生クラスからアクセス可能
 - ▶ virtual/override: 派生クラスでメソッドの振る舞いを上書き可能にする
 - ▶ abstract: 抽象メンバー/クラス。派生クラスで必ず実装する必要がある
 - ▶ sealed: それ以上継承不可にする

継承 (ii)

例:

```
using System;

public class Animal // 基底クラス
{
    protected string name;
    public Animal(string name)
    {
        this.name = name;
    }

    public virtual void Speak()
    {
        Console.WriteLine($"{name} says: ...?"); // デフォルト
    }
}
```

継承 (iii)

```
// 派生クラス
public class Dog : Animal
{
    public Dog(string name) : base(name) { }

    // 基底の振る舞いを上書き
    public override void Speak()
    {
        Console.WriteLine($"{name} says: Woof!");
    }
}
```

- Animal 型を継承した Dog class.
- Speak()を上書き

継承 (iv)

```
public class Cat : Animal
{
    public Cat(string name) : base(name) { }

    public override void Speak()
    {
        Console.WriteLine($"{name} says: Meow!");
    }
}
```

- Animal 型を継承した Cat class.
- Speak() を上書き

継承 (v)

```
public class Main
{
    public static void Main()
    {
        Animal a1 = new Dog("Pochi");
        Animal a2 = new Cat("Tama");

        // ポリモーフィズム：参照型はAnimalだが
        // 実体の型に応じて適切なSpeakが呼ばれる
        a1.Speak(); // Pochi says: Woof!
        a2.Speak(); // Tama says: Meow!
    }
}
```

- コンストラクタで `base(...)` を使うと、基底クラスのコンストラクタを呼べる
- 抽象クラス・メソッドの例:

継承 (vi)

```
public abstract class Shape
{
    public abstract double Area(); // 派生クラスで必須実装
}
public class Rectangle : Shape
{
    public double Width { get; }
    public double Height { get; }
    public Rectangle(double w, double h)
    {
        Width = w;
        Height = h;
    }

    public override double Area() => Width * Height;
}
```

継承 (vii)

- sealed クラス(または sealed メソッド)はそれ以上の継承や上書きを禁止:

```
public sealed class FinalClass { }
```

- 参考: インターフェイスは継承ではなく「契約」
 - public interface IRunner { void Run(); }
 - クラスは: class Person : IRunner { public void Run() { } }

文字列とコレクションとジェネリック型

文字列

- C では文字列を扱う際は `char[]` として扱うか、`char*` として扱うかの 2 択。
- C# では `string` や `char[]` を使うのが一般的。
- 文字列は `string` 型で扱うのが一般的。
- また、`$"{...}"` で文字列をフォーマットすることも可能。

```
string str = "Hello";  
char[] chars = { 'H', 'e', 'l', 'l', 'o' };  
  
string format = $"Hello, {str}!"; // => "Hello, Hello!"  
// printf("Hello, %s", str);と同じ感じ
```

コレクション

- C では `int[]` や `int*` を使うのが一般的。
- C# では `List<int>` や `int[]` を使うのが一般的。

C:

```
int ints[] = { 1, 2, 3, }
```

C#:

```
int[] ints = { 1, 2, 3 };
```

```
//or
```

```
List<int> list = new List<int> { 1, 2, 3 };
```

コレクション (ii)

C:

```
for (int i = 0; i < 3; i++) {  
    printf("%d ", ints[i]);  
}
```

C#:

```
foreach (int i in ints)  
{  
    Console.Write($"{i} ");  
}
```

コレクション (iii)

- ここでは List の型を使って説明したが、他にも同じように使える型もある。

Array	固定の長さ。インデックスでアクセスできる	T[]
List	可変長。インデックスでアクセスできる	List<T>
Dictionary	キーと値のペアでアクセスできる	Dictionary<TKey, TValue>
HashSet	重複しない要素を保持するコレクション	HashSet<T>
Stack	後入れ先出しのデータ構造	Stack<T>
Queue	先入れ先出しのデータ構造	Queue<T>

ジェネリック型

- ジェネリック型は、型パラメータを指定することで、型の柔軟性を高めることができる。
- 今までのスライドで `List<T>` を使った例を紹介した。
- `T` に `int` や `double` などの型を指定することで、柔軟に型を指定できる。
- 正直一度使ってみないと、どういう感じに役に立つのかわからないと思われる。
- そんなのがあるんだ～程度で良い。

C 言語とは異なるコードスタイル

無駄に気持ち悪い波括弧

- C 言語では、波括弧のために行を用意することはない。

例:

```
for (int i = 0; i < 3; i++) {  
    printf("%d ", i);  
}
```

- しかし、C# では波括弧のために行を用意して書いてやるのが一般的なフォーマットである。

例:

```
for (int i = 0; i < 3; i++)  
{  
    Console.Write($"{i} ");  
}
```

構造体以外にも関数の一文字目を大文字にする。

```
public static void Main()  
{  
    Console.WriteLine("Hello World!");  
}
```

- **ただ、定義する変数名の一文字目は小文字なので注意**

これ以外にもいろいろあるが、正直きれいに読めれば大丈夫であるため、過度に気にする必要はない。

ここで紹介したものは **Microsoft Docs Lean .NET C# 一般的なコード規則**で詳細に書かれているため、暇な人は一読してみてもいいだろう。