NẠP CHỒNG TOÁN TỬ

Nạp chồng toán tử

Khái niệm nạp chồng toán tử

Đối với các kiểu dữ liệu số, C# định nghĩa sẵn một số phép toán như các phép toán số học, phép toán so sánh, phép toán tăng giảm. Đối với kiểu string, như chúng ta đã biết, được định sẵn phép toán cộng xâu.

Tuy nhiên, các kiểu dữ liệu (class) do người dùng định nghĩa lại không thể sử dụng ngay các phép toán đó được.

Ví dụ, nếu người dùng định nghĩa kiểu số phức, các phép toán cơ bản trên kiểu số phức lại không thể thực hiện được ngay, mặc dù về mặt toán học các phép toán đối với số phức không có gì khác biệt với kiểu số được C# định nghĩa.

Để giải quyết những vấn đề tương tự, C# cho phép *nạp chồng toán tử*, tức là cho phép định nghĩa lại những phép toán đã có với các kiểu dữ liệu do người dùng xây dựng.

Nạp chồng phương thức (method overloading) cùng với *nạp chồng toán tử* (operator overloading) là hai hiện tượng thuộc về nguyên lý *đa hình tĩnh* (static polymorphism).

Cách nạp chồng toán tử

Hãy cùng thực hiện và phân tích ví dụ sau để hiểu cách nạp chồng toán tử. Chú ý xem xét cú pháp nạp chồng đối với mỗi toán tử.

```
using System;
namespace P01_OperatorOverload
{
    /// <summary>
    /// lớp biểu diễn hình hộp
    /// </summary>
    internal class Box
    {
        public double Length { get; set; }
        public double Breadth { get; set; }
        public double Height { get; set; }
        public Box() { }
        public Box(double length, double breadth, double height)
        {
            Length = length;
            Breadth = breadth;
            Height = height;
        }
        /// <summary>
        /// tính thể tích khối hộp
        /// </summary>
```

```
        public double Volume => Length * Breadth * Height;
        // nạp chồng phép cộng
        public static Box operator +(Box b, Box c)
        {
            Box box = new Box
            {
                Length = b.Length + c.Length,
                Breadth = b.Breadth + c.Breadth,
                Height = b.Height + c.Height
            };
            return box;
        }
        // nạp chồng phép so sánh bằng
        public static bool operator ==(Box lhs, Box rhs)
        {
            bool status = false;
            if (lhs.Length == rhs.Length && lhs.Height == rhs.Height
               && lhs.Breadth == rhs.Breadth)
            {
                status = true;
            }
            return status;
        }
        // nạp chồng phép so sánh khác
        public static bool operator !=(Box lhs, Box rhs)
        {
            bool status = false;
            if (lhs.Length != rhs.Length || lhs.Height != rhs.Height ||
               lhs.Breadth != rhs.Breadth)
            {
                status = true;
            }
            return status;
        }
        // nạp chồng phép so sánh nhỏ hơn
        public static bool operator <(Box lhs, Box rhs)
        {
            bool status = false;
            if (lhs.Length < rhs.Length && lhs.Height < rhs.Height
               && lhs.Breadth < rhs.Breadth)
            {
                status = true;
            }
            return status;
        }
        // nạp chồng phép so sánh lớn hơn
        public static bool operator >(Box lhs, Box rhs)
        {
            bool status = false;
            if (lhs.Length > rhs.Length && lhs.Height >
               rhs.Height && lhs.Breadth > rhs.Breadth)
```

```
            {
                status = true;
            }
            return status;
        }
        public override string ToString()
        {
            return string.Format("({0}, {1}, {2})", Length, Breadth, Height);
        }
    }
    internal class Program
    {
        private static void Main(string[] args)
        {
            Box Box1 = new Box(6, 7, 5);
            Box Box2 = new Box(12, 13, 10);
            Box Box3 = new Box();
            Box Box4 = new Box();
            /* phép cộng hai hình hộp cho ra hình hộp khác có kích thước
             * bằng tổng kích thước của hai hộp */
            Box3 = Box1 + Box2;
            Console.WriteLine("Box 3: {0}", Box3.ToString());
            Console.WriteLine("Volume of Box3 : {0}", Box3.Volume);
            // so sánh hai hình hộp
            if (Box1 > Box2)
                Console.WriteLine("Box1 lớn hơn Box2");
            else
                Console.WriteLine("Box1 không lớn hơn Box2");
            if (Box3 == Box4)
                Console.WriteLine("Box3 bằng Box4");
            else
                Console.WriteLine("Box3 không bằng Box4");
            Console.ReadKey();
        }
    }
}
```

Trong ví dụ trên chúng ta đã thực hiện nạp chồng cho phép toán cộng (+), các phép so sánh (bằng ==, khác !=, lớn hơn >, nhỏ hơn <).

Cú pháp khai báo này được tổng hợp lại dưới đây:

```
public static Box operator +(Box b, Box c) {...}
public static bool operator ==(Box lhs, Box rhs) {...}
public static bool operator !=(Box lhs, Box rhs) {...}
public static bool operator <(Box lhs, Box rhs) {...}
public static bool operator >(Box lhs, Box rhs) {...}
```

Nếu để ý kỹ hơn nữa chúng ta thấy, đây đều là các phép toán binary. Cách nạp chồng các phép toán này có cùng một cú pháp.

Mỗi loại phép toán sẽ có cách nạp chồng riêng. Tuy nhiên, cú pháp chung là

```
public static <return_type> operator <operator>(<parameters>) { ... }
```

Các toán tử có thể nạp chồng: `+, -, !, ~, ++, --, +, -, *, /, %, ==, !=, <, >, <=, >=`

Ngoài ra phép toán indexer cũng là một phép toán có thể nạp chồng.

Một số lưu ý khi nạp thực hiện chồng toán tử

Các phép toán chia làm ba loại:

- *unary* (chỉ cần một toán hạng, như phép toán tăng `++`, phép toán giảm `--`)
- *binary* (cần hai toán hạng, như các phép toán `+`, `-`, `*`, `/`)
- *ternary* (cần ba toán hạng, như phép toán điều kiện `?`)

Do đó, khi nạp chồng phép toán nào thì phải cung cấp đủ lượng tham số phù hợp. Ví dụ, khi nạp chồng phép toán binary (như `+`, `-`) thì phải cấp 2 tham số như đã làm ở trên.

Phép toán tăng giảm (`++`, `--`) thuộc loại unary nên trong danh sách tham số chỉ cần 1 tham số. Các phép toán này cũng không có giới hạn gì khi nạp chồng. Cùng ví dụ với lớp Box trên:

```
public static Box operator ++ (Box b)
{
    return new Box(b.Length++, b.Breadth++, b.Height++);
}
```

Các phép toán số học (`+, - *, /, %`) không đặt ra giới hạn gì khi nạp chồng. Bạn chỉ cần tuân thủ đúng cú pháp như trên là được.

Bạn thậm chí có thể nạp chồng cùng một phép toán nhiều lần. Ví dụ, bạn hoàn toàn có thể thêm nạp chồng phép `+` một lần nữa như sau:

```
public static Box operator +(Box b, double size)
{
    return new Box(b.Length += size, b.Breadth += size, b.Height + size);
}
```

Ở đây bạn đã nạp chồng phép cộng Box với một số thực. Điều kiện để nạp chồng phép toán nhiều lần là danh sách tham số của mỗi lần nạp chồng phải khác nhau.

Đối với các phép toán so sánh, bạn phải thực hiện nạp chồng cả cặp. Nghĩa là, nếu nạp chồng phép so sánh bằng `==` thì đồng thời phải nạp chồng cả phép so sánh khác `!=`; nếu nạp chồng phép so sánh hơn `>` thì phải nạp chồng cả phép so sánh kém `<`.

Các phép gán (`+=`, `-=`, v.v.) không cho phép nạp chồng trực tiếp. Tuy nhiên, nếu bạn đã nạp chồng phép `+`, `-`, v.v. thì các phép toán này tự nhiên sẽ được nạp chồng. Ví dụ, nếu bạn đã nạp chồng phép cộng Box với 1 số như trên thì hoàn toàn có thể gọi lệnh

```
var Box5 = Box4 += 5; // phép cộng gán với số
```

Riêng phép toán indexer có cách thực hiện nạp chồng riêng dưới đây.

Nạp chồng phép toán indexer

Indexer là một phép toán giúp client code sử dụng object tương tự như khi sử dụng mảng. Indexer thường được sử dụng với với các kiểu dữ liệu chứa trong nó một tập hợp dữ liệu (collection hoặc array). Indexer giúp đơn giản hóa việc sử dụng ở client code.

Trước khi xem cú pháp nạp chồng toán tử indexer, hãy cùng thực hiện ví dụ sau đây:

```
namespace P02_IndexerOverload
{
    using static System.Console;
    class Program
    {
        static void Main(string[] args)
        {
            var vector1 = new Vector(1, 2, 3);
            WriteLine($"vector 1: {vector1}");
            ReadLine();
        }
    }
    class Vector
    {
        private double[] _components;
        public Vector(int dimension)
        {
            _components = new double[dimension];
        }
        public Vector(params double[] components)
        {
            _components = components;
        }
        public override string ToString()
        {
            return $"({string.Join(", ", _components)})";
        }
    }
}
```

Ví dụ này xây dựng một `class Vector` đơn giản dành cho vector n-chiều. Cả vector được lưu trong một mảng `private _components` (mỗi phần tử của mảng là kích thước một chiều của vector). Class này có 2 overload cho constructor, một cái nhận số chiều làm tham số, một cái nhận mảng `double` làm tham số.

Bạn có muốn truy xuất giá trị từng chiều của vector này như truy xuất phần tử của mảng không? Tức là viết kiểu `vector[0]`, `vector[1]`, v.v., trong đó 0, 1, là chỉ số chiều.

Cú pháp nạp chồng indexer

Cú pháp nạp chồng phép toán indexer cho class gần giống property, trong đó phải có ít nhất một trong hai phương thức `get`/`set`, dùng để trả lại giá trị và gán giá trị. Khác biệt duy nhất ở chỗ indexer bắt buộc sử dụng từ khóa `this` với cặp dấu ngoặc vuông. Biến làm khóa phải đặt trong cặp dấu ngoặc vuông.

```
public TValue this[TKey key]
{
    get{ }
    set{ }
}
```

Trong đó:

1. `TValue` là kiểu dữ liệu trả về, `TKey` là kiểu dữ liệu của khóa;
2. số lượng khóa có thể nhiều hơn 1;
3. kiểu của khóa có thể là bất kỳ kiểu dữ liệu nào (không nhất thiết là số hoặc chuỗi);
4. phương thức `get` và `set` hoạt động giống như đối với thuộc tính.

Nạp chồng toán tử indexer cho lớp Vector

Thêm đoạn code sau vào class Vector:

```
public double this[int index]
{
    get => (index < _components.Length) ? _components[index] : double.NaN;
    set { if (index < _components.Length) _components[index] = value; }
}
```

Cấu trúc này nhìn giống hệt như full property, ngoại trừ tên gọi `this[int index]`. Đoạn code này đã thực hiện nạp chồng toán tử indexer cho lớp Vector.

Logic hoạt động của getter rất đơn giản. Nếu index nhỏ hơn số phần tử của mảng thì trả lại giá trị tương ứng index, nếu không thì trả về giá trị NaN (Not a Number). Setter chỉ gán giá trị cho phần tử tương ứng của mảng.

Nếu phương thức `get`/`set` chỉ chứa một lệnh duy nhất có thể sử dụng cú pháp "expression body" cho ngắn gọn. Trong code của getter ở trên chúng ta đã sử dụng cấu trúc này.

Expression body là một lối viết xuất hiện từ C# 6: nếu thân của phương thức chỉ chứa một lệnh duy nhất có thể sử dụng cấu trúc như sau để viết:

```
Tên_phương_thức() => lệnh;
```

Từ C# 7 có thể sử dụng expression body cho cả phương thức get và set của property.

Bây giờ bạn có thể truy xuất từng chiều của vector như sau:

```
static void Main(string[] args)
{
    var vector1 = new Vector(1, 2, 3);
    WriteLine($"vector 1: {vector1}");
    var x = vector1[0];
    var y = vector1[1];
    var z = vector1[2];
    WriteLine($"Vector components: x = {x}, y = {y}, z = {z}");
    vector1[2] = 30;
    vector1[1] = 20;
    vector1[0] = 10;
    WriteLine($"vector 1: {vector1}");
    ReadLine();
}
```

Dưới đây là code đầy đủ của ví dụ trên:

```
namespace P02_IndexerOverload
{
    using static System.Console;
    class Program
    {
        static void Main(string[] args)
        {
            var vector1 = new Vector(1, 2, 3);
            WriteLine($"vector 1: {vector1}");
            var x = vector1[0];
            var y = vector1[1];
            var z = vector1[2];
            WriteLine($"Vector components: x = {x}, y = {y}, z = {z}");
            vector1[2] = 30;
            vector1[1] = 20;
            vector1[0] = 10;
            WriteLine($"vector 1: {vector1}");
            ReadLine();
        }
    }
    class Vector
    {
        public double this[int index]
        {
            get => (index < _components.Length) ? _components[index] : double.NaN;
            set { if (index < _components.Length) _components[index] = value; }
```

```
        }
        private double[] _components;
        public Vector(int dimension)
        {
            _components = new double[dimension];
        }
        public Vector(params double[] components)
        {
            _components = components;
        }
        public override string ToString()
        {
            return $"({string.Join(", ", _components)})";
        }
    }
}
```