ГЛАВА 9

Специальные приемы построения типов

В этой главе вы расширите горизонты вашего понимания языка C#, рассмотрев ряд более сложных (но весьма полезных) синтаксических конструкций. Сначала мы с вами выясним, как использовать метод индексатора. Этот механизм в C# позволяет строить пользовательские типы, обеспечивающие доступ к внутренним подтипам на основе синтаксиса массивов. Научившись строить методы индексатора, вы затем узнаете, как перегружать различные операции (+, –,<, > и т.д.) и явно или неявно создавать пользовательские подпрограммы преобразования типов (а также узнаете, зачем это может понадобиться).

Во второй половине главы будет рассмотрен небольшой набор ключевых слов C#, которые позволяют реализовать весьма интересные конструкции, хотя используются не очень часто. Вы узнаете о том, как с помощью ключевых слов `checked` и `unchecked` программно учитывать условия переполнения и потери значимости, а также о том, как создается "небезопасный" программный контекст, обеспечивающий возможность непосредственного управления ссылочными типами в C#. Завершается глава обсуждением роли директив препроцессора C#.

Создание пользовательских индексаторов

Как программисты, мы прекрасно знаем, что с помощью индексов можно получить доступ к отдельным элементам, содержащимся в стандартном массиве.

```
// Объявление массива целых значений.
int[] myInts = { 10, 9, 100, 432, 9874};

// Использование операции [] для доступа к элементам.
for(int j = 0; j < myInts.Length; j++)
    Console.WriteLine("Индекс {0} = {1} ", j, myInts[j]);
```

Этот программный код ни в коем случае не претендует на новизну. Но язык C# дает возможность строить пользовательские классы и структуры, которые могут индексироваться подобно стандартным массивам. Поэтому совсем не удивительно, что метод, который обеспечивает такой доступ к элементам, называется *индексатором*.

Перед тем как приступить к созданию соответствующей конструкции, мы рассмотрим один пример. Предположим, что поддержка метода индексатора уже до-

бавлена в пользовательскую коллекцию Garage (гараж), уже рассматривавшуюся в главе 8. Проанализируйте следующий пример ее использования.

```
// Индексаторы обеспечивают доступ к элементам подобно массивам.
public class Program
{
   static void Main(string[] args)
   {
      Console.WriteLine("***** Забавы с индексаторами *****\n");

      // Предположим, что Garage имеет метод индексатора.
      Garage carLot = new Garage();

      // Добавление в гараж машин с помощью индексатора.
      carLot[0] = new Car("FeeFee", 200);
      carLot[1] = new Car("Clunker", 90);
      carLot[2] = new Car("Zippy", 30);

      // Чтение и отображение элементов с помощью индексатора.
      for (int i = 0; i < 3; i++)
      {
         Console.WriteLine("Номер машины: {0}", i);
         Console.WriteLine("Название: {0}", carLot[i].PetName);
         Console.WriteLine("Максимальная скорость: {0}",
            carLot[i].CurrSpeed);
         Console.WriteLine();
      }
      Console.ReadLine();
   }
}
```

Как видите, индексаторы ведут себя во многом подобно пользовательской коллекции, поддерживающей интерфейсы IEnumerator и IEnumerable. Основное различие в том, что вместо доступа к содержимому посредством типов интерфейса вы можете работать с внутренней коллекцией автомобилей, как с обычным массивом.

Здесь возникает вопрос: "Как сконфигурировать класс (или структуру), чтобы обеспечить поддержку соответствующих функциональных возможностей?" Индексатор в C# представляет собой несколько "искаженное" свойство. Для создания индексатора в самой простой форме используется синтаксис this[]. Вот как может выглядеть подходящая модификация типа Garage.

```
// Добавление индексатора в определение класса.
public class Garage : IEnumerable    // для каждого элемента
{
...
   // Использование ArrayList для типов Car.
   private ArrayList carArray = new ArrayList();

   // Индексатор возвращает тип Car, соответствующий
   // числовому индексу.
   public Car this[int pos]
   {
      // ArrayList тоже имеет индексатор!
```

```
        get { return (Car)carArray[pos]; }
        set {carArray.Add(value);}
    }
}
```

Если не обращать внимания на ключевое слово `this`, то объявление индексатора очень похоже на объявление свойства в C#. Но следует подчеркнуть, что индексаторы не обеспечивают иных функциональных возможностей массива, кроме возможности использования операции индексирования. Другими словами, пользователь объекта не может применить программный код, подобный следующему.

```
// Используется свойство ArrayList.Count? Нет!
Console.WriteLine("Машин в наличии: {0} ", carLot.Count);
```

Для поддержки этой функциональной возможности вы должны добавить свое свойство `Count` в тип `Garage` и, соответственно, делегат.

```
public class Garage: IEnumerable
{
    ...
    // Локализация/делегирование в действии снова.
    public int Count { get { return carArray.Count; } }
}
```

Итак, индексаторы — это еще одна синтаксическая "конфетка", поскольку соответствующих функциональных возможностей можно достичь и с помощью "обычных" методов. Например, если бы тип `Garage` не поддерживал индексатор, все равно можно было бы позволить "внешнему миру" взаимодействовать с внутренним массивом, используя для этого именованное свойство или традиционные методы чтения и модификации данных (accessor/mutator). Но при использовании индексаторов пользовательские типы коллекции лучше согласуются со структурой библиотек базовых классов .NET.

Исходный код. Проект SimpleIndexer размещен в подкаталоге, соответствующем главе 9.

Вариации индексатора для типа Garage

В своем текущем виде тип `Garage` определяет индексатор, который позволяет вызывающей стороне идентифицировать внутренние элементы, используя числовое значение. Но это не является непременным требованием метода индексатора. Предположим, что объекты `Car` содержатся в `System.Collections.Specialized.ListDictionary`, а не в `ArrayList`. Поскольку типы `ListDictionary` позволяют доступ к содержащимся типам с помощью ключевых маркеров (таких как, например, строки), можно создать новый индексатор `Garage`, подобный показанному ниже.

```
public class Garage : IEnumerable
{
    private ListDictionary carDictionary = new ListDictionary();

    // Этот индексатор возвращает соответствующий тип Car
    // на основе строкового индекса.
    public Car this[string name]
```

```
    {
        get { return (Car)carDictionary[name]; }
        set { carDictionary[name] = value; }
    }

    public int Length { get { return carDictionary.Count; } }

    public IEnumerator GetEnumerator()
    { return carDictionary.GetEnumerator(); }
}
```

Вызывающая сторона теперь может взаимодействовать с машинами внутри так, как показано ниже.

```
public class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("***** Забавы с индексаторами *****\n");
        Garage carLot = new Garage();

        // Добавление именованных машин в гараж.
        carLot["FeeFee"] = new Car("FeeFee", 200, 0);
        carLot["Clunker"] = new Car("Clunker", 90, 0);
        carLot["Zippy"]  = new Car("Zippy", 30, 0);

        // Доступ к Zippy.
        Car zippy = carLot["Zippy"];
        Console.WriteLine("{0} едет со скоростью {1} км/ч",
            zippy.PetName, zippy.CurrSpeed);
        Console.ReadLine();
    }
}
```

Индексаторы могут быть и перегруженными. Так, чтобы позволить вызывающей стороне доступ к внутренним элементам посредством числового индекса *или* строковых значений, вы можете определить множество индексаторов для одного типа.

Исходный код. Проект StringIndexer размещен в подкаталоге, соответствующем главе 9.

Внутреннее представление индексаторов типов

Мы рассмотрели примеры метода индексатора в C#, и пришло время выяснить, как представляются индексаторы в терминах CIL. Если открыть числовой индексатор типа `Garage`, то будет видно, что компилятор C# создает свойство `Item`, которое сводится к подходящей паре методов `get/set`.

```
property instance class SimpleIndexer.Car Item(int32)
{
    .get instance class SimpleIndexer.Car SimpleIndexer.Garage::get_Item(int32)
    .set instance void SimpleIndexer.Garage::set_Item(int32,
        class SimpleIndexer.Car)
} // end of property Garage::Item
```

Методы get_Item() и set_Item() будут реализованы аналогично любому другому свойству .NET, например:

```
method public hidebysig specialname instance class SimpleIndexer.Car
get_Item(int32 pos) cil managed
{
   Code size    22 (0x16)
   .maxstack 2
   .locals init ([0] class SimpleIndexer.Car CS$1$0000)
   IL_0000: ldarg.0
   IL_0001: ldfld class [mscorlib]
      System.Collections.ArrayList SimpleIndexer.Garage::carArray
   IL_0006: ldarg.1
   IL_0007: callvirt instance object [mscorlib]
     System.Collections.ArrayList::get_Item(int32)
   IL_000c: castclass SimpleIndexer.Car
   IL_0011: stloc.0
   IL_0012: br.s IL_0014
   IL_0014: ldloc.0
   IL_0015: ret
} // end of method Garage::get_Item
```

Заключительные замечания об индексаторах

Чтобы получить настоящую экзотику, вы можете создать индексатор, который имеет множество параметров. Предположим, что у нас есть пользовательская коллекция, которая хранит элементы в двумерном массиве. В этом случае вы можете создать метод индексатора, показанный ниже.

```
public class SomeContainer
{
   private int[,] my2DintArray = new int[10, 10];

   public int this[int row, int column]
   {  /* прочитать или установить значение 2D-массива */  }
}
```

В заключение следует заметить, что индексаторы могут определяться и для типа интерфейса .NET, что обеспечивает реализующим интерфейс типам возможность его настройки. Вот пример такого интерфейса.

```
public interface IEstablishSubObjects
{
   // Этот интерфейс определяет индексатор, возвращающий
   // строки на основе числового индекса.
   string this[int index] { get; set; }
}
```

Пожалуй, об индексаторах C# уже сказано достаточно. Перейдем к рассмотрению еще одного подхода, используемого в некоторых (но не во всех) языках программирования .NET: это перегрузка операций.

Перегрузка операций

В C#, как и в любом другом языке программирования, есть свой ограниченный набор лексем, используемых для выполнения базовых операций со встроенными типами. Так, вы знаете, что операция + применима к двум целым числам и в результате дает их сумму.

```
// Операция + с целыми числами.
int a = 100;
int b = 240;
int c = a + b;    // c теперь равно 340
```

Снова заметим, что это не новость, но вы, наверное, заметили и то, что одна и та же операция + может применяться к большинству встроенных типов данных C#. Рассмотрите, например, следующий фрагмент программного кода.

```
// Операция + со строками.
string s1 = "Hello";
string s2 = " world!";
string s3 = s1 + s2;    // s3 теперь равно "Hello world!"
```

По сути, операция + функционирует уникальным образом в зависимости от поставляемых типов данных (в данном случае это строки или целые числа). Когда операция + применяется к числовым типам, результатом является сумма операндов, а когда операция + применяется к строковым типам, результатом будет конкатенация строк.

Язык C# обеспечивает возможность построения пользовательских классов и структур, которые будут по-своему отвечать на один и тот же набор базовых лексем (таких, как операция +). При этом следует заметить, что можно "перегружать" не все встроенные операции C#. В табл. 9.1 указаны возможности перегрузки базовых операций.

Таблица 9.1. Возможности перегрузки операций

Операции C#	Возможность перегрузки
+, –, !, ~, ++, --, true, false	Эти унарные операции допускают перегрузку
+, –, *, /, %, &, |, ^, <<, >>	Эти бинарные операции допускают перегрузку
==, !=, <, >, <=, >=	Операции сравнения допускают перегрузку. В C# требуется, чтобы перегрузка "родственных" операций (т.е. < и >, <= и >=, == и !=) выполнялась одновременно
[]	Операция [] не допускает перегрузку. Но, как было показано выше, аналогичные перегрузке возможности обеспечивает конструкция индексатора
()	Операция () не допускает перегрузку. Но, как будет показано ниже, аналогичные перегрузке возможности обеспечивают пользовательские методы преобразования
+=, –=, *=, /=, %=, &=, |=, ^=, <<=, >>=	Операторные сокращения с присваиванием сами по себе не допускают перегрузку, однако для них перегруженная форма получается автоматически в результате перегрузки соответствующей бинарной операции

Перегрузка бинарных операций

Чтобы проиллюстрировать процесс перегрузки бинарных операций, рассмотрим следующую простую структуру `Point` (точка).

```
// Самая обычная структура C#.
public struct Point
{
   private int x, y;
   public Point(int xPos, int yPos)
   {
      x = xPos;
      y = yPos;
   }

   public override string ToString()
   {
      return string.Format("[{0}, {1}]", this.x, this.y);
   }
}
```

Можно, скажем, рассмотреть сложение типов `Point`. Можно также вычесть один тип `Point` из другого. Например, вы можете записать следующий программный код.

```
// Сложение и вычитание двух точек.
static void Main(string[] args)
{
   Console.WriteLine("*** Забавы с перегруженными операциями ***\n");

   // Создание двух точек.
   Point ptOne = new Point(100, 100);
   Point ptTwo = new Point(40, 40);
   Console.WriteLine("ptOne = {0}", ptOne);
   Console.WriteLine("ptTwo = {0}", ptTwo);

   // Сложение точек в одну большую точку?
   Console.WriteLine("ptOne + ptTwo: {0} ", ptOne + ptTwo);

   // Вычитание одной точки из другой дает меньшую точку?
   Console.WriteLine("ptOne - ptTwo: {0} ", ptOne - ptTwo);
   Console.ReadLine();
}
```

Чтобы позволить пользовательскому типу по-своему отвечать на встроенные операции, в C# предлагается ключевое слово `operator`, которое можно использовать только со *статическими* методами. Перегруженной бинарной операции (такой, как + или –) на вход подаются два аргумента, которые имеют тип определяющего класса (в данном примере это `Point`), как показывает следующий программный код.

```
// Более 'интеллектуальный' тип Point.
public struct Point
{
...
```

```
    // перегруженная операция +
    public static Point operator + (Point p1, Point p2)
    { return new Point(p1.x + p2.x, p1.y + p2.y); }

    // перегруженная операция -
    public static Point operator - (Point p1, Point p2)
    { return new Point(p1.x - p2.x, p1.y - p2.y); }
}
```

По логике операция + должна просто возвратить совершенно новый объект `Point`, полученный в результате суммирования соответствующих полей входных параметров `Point`. Поэтому, когда вы пишете `pt1 + pt2`, можете представлять себе следующий скрытый вызов статического метода операции +.

```
// p3 = Point.операция+ (p1, p2)
p3 = p1 + p2;
```

Точно так же `p1 - p2` отображается в следующее.

```
// p3 = Point.операция- (p1, p2)
p3 = p1 - p2;
```

Операции += и –+

Если вы изучаете C#, уже имея опыт использования C++, то можете обратить внимание на отсутствие возможности перегрузки операторных сокращений, включающих операцию присваивания (+=, -= и т.д.). Не волнуйтесь, в C# операторные сокращения с присваиванием моделируются автоматически, если тип предполагает перегрузку соответствующей бинарной операции. Поэтому, поскольку структура `Point` уже использует перегрузку операций + и –, вы можете записать следующее.

```
// Перегрузка бинарных операций автоматически влечет перегрузку
// операторных сокращений с присваиванием.
static void Main(string[] args)
{
    // Автоматическая перегрузка +=
    Point ptThree = new Point(90, 5);
    Console.WriteLine("ptThree = {0}", ptThree);
    Console.WriteLine("ptThree += ptTwo: {0}", ptThree += ptTwo);

    // Автоматическая перегрузка -=
    Point ptFour = new Point(0, 500);
    Console.WriteLine("ptFour = {0}", ptFour);
    Console.WriteLine("ptFour -= ptThree: {0}", ptFour -= ptThree);
}
```

Перегрузка унарных операций

В C# также позволяется перегрузка унарных операций, таких как, например, ++ и --. При перегрузке унарной операции вы тоже должны с помощью ключевого слова `operator` определить статический метод, но в данном случае передается только один параметр, который должен иметь тип, соответствующий определяю-

щему классу или структуре. Например, если добавить в Point следующие перегруженные операции

```
public struct Point
{
...
   // Добавление 1 к поступившему Point.
   public static Point operator ++(Point p1)
   { return new Point(p1.x+1, p1.y+1); }

   // Вычитание 1 от поступившего Point.
   public static Point operator --(Point p1)
   { return new Point(p1.x-1, p1.y-1); }
}
```

то вы получите возможность увеличивать или уменьшать на единицу значения X и Y объекта Point, как показано ниже.

```
static void Main(string[] args)
{
...
   // Применение унарных операций ++ и -- к Point.
   Console.WriteLine("++ptFive = {0}", ++ptFive);
   Console.WriteLine("--ptFive = {0}", --ptFive);
}
```

Перегрузка операций проверки на тождественность

Вы можете помнить из материала главы 3, что System.Object.Equals() можно переопределить, чтобы сравнение типов выполнялось на основе значений (а не ссылок). Если вы переопределите Equals() (и связанный с Equals() метод System.Object.GetHashCode()), то будет очень просто задать перегрузку операций проверки на тождественность (== и !=). Для иллюстрации мы рассмотрим обновленный тип Point.

```
// Такая 'инкарнация' Point задает также перегрузку операций == и !=.
public struct Point
{
...
   public override bool Equals(object o)
   {
      if(o is Point)
      {
         if( ((Point)o).x == this.x &&
             ((Point)o).y == this.y )
            return true;
      }
      return false;
   }
```

```
    public override int GetHashCode()
    { return this.ToString().GetHashCode(); }

    // Здесь позволяется перегрузка операций == и !=.
    public static bool operator ==(Point p1, Point p2)
    { return p1.Equals(p2); }

    public static bool operator !=(Point p1, Point p2)
    { return !p1.Equals(p2); }
}
```

Обратите внимание на то, что данная реализация операций == и != просто вызывает переопределенный метод Equals(), который и выполняет основную работу. С учетом этого вы можете теперь использовать свой класс Point так.

```
// Использование перегруженных операций проверки на тождественность.
static void Main(string[] args)
{
...
    Console.WriteLine("ptOne == ptTwo : {0}", ptOne == ptTwo);
    Console.WriteLine("ptOne != ptTwo : {0}", ptOne != ptTwo);
}
```

Как видите, здесь два объекта сравниваются с помощью операций == и !=, а не с помощью "менее естественного" вызова Object.Equals(). При использовании перегрузки операций проверки на тождественность для класса имейте в виду, что в C# требуется, чтобы при переопределении операции == *обязательно* переопределялась и операция != (если вы забудете это сделать, компилятор вам напомнит).

Перегрузка операций сравнения

Из материала главы 7 вы узнали о том, как реализовать интерфейс IComparable, чтобы иметь возможность сравнения подобных объектов. В дополнение к этому для того же класса вы можете использовать перегрузку операций сравнения (<, >, <= и >=). Подобно операциям проверки на тождественность, в C# требуется, чтобы при перегрузке < выполнялась и перегрузка >. Это же касается и операций <= и >=. Если тип Point использует перегрузку операций сравнения, пользователь объекта получает возможность сравнивать объекты Point так, как показано ниже.

```
// Использование перегруженных операций < и >.
static void Main(string[] args)
{
...
    Console.WriteLine("ptOne < ptTwo : {0}", ptOne < ptTwo);
    Console.WriteLine("ptOne > ptTwo : {0}", ptOne > ptTwo);
}
```

В предположении о том, что интерфейс IComparable реализован, перегрузка операций сравнения оказывается тривиальной. Вот как может выглядеть обновленное определение класса.

```
// Можно сравнивать объекты Point с помощью операций сравнения.
public struct Point : IComparable
```

```
{
...
   public int CompareTo(object obj)
   {
      if (obj is Point)
      {
         Point p = (Point)obj;
         if (this.x > p.x && this.y > p.y)
            return 1;
         if (this.x < p.x && this.y < p.y)
            return -1;
         else
            return 0;
      }
      else
         throw new ArgumentException();
   }

   public static bool operator <(Point p1, Point p2)
   { return (p1.CompareTo(p2) < 0); }

   public static bool operator >(Point p1, Point p2)
   { return (p1.CompareTo(p2) > 0); }

   public static bool operator <=(Point p1, Point p2)
   { return (p1.CompareTo(p2) <= 0); }

   public static bool operator >=(Point p1, Point p2)
   { return (p1.CompareTo(p2) >= 0); }
}
```

Внутреннее представление перегруженных операций

Подобно любому элементу программы C#, перегруженные операции представляются специальными элементами синтаксиса CIL. Откройте, например, компоновочный блок `OverloadedOps.exe` с помощью `ildasm.exe`. Как показано на рис. 9.1, перегруженные операции внутри блока представляются скрытыми методами (это, например, `op_Addition()`, `op_Subtraction()`, `op_Equality()` и т.д.).

Теперь, если рассмотреть CIL-инструкции для метода `op _ Addition`, то вы обнаружите, что `csc.exe` добавляет в метод ключевое слово `specialname`.

```
.method public hidebysig specialname static
   valuetype OverloadedOps.Point
      op_Addition(valuetype OverloadedsOps.Point p1,
         valuetype OverloadedOps.Point p2) cil managed
{
...
}
```

Рис. 9.1. В терминах CIL перегруженные операции отображаются в скрытые методы

Итак, любая операция, допускающая перегрузку, сводится в терминах CIL к специальному именованному методу. В табл. 9.2 раскрывается соответствие имен типичных операций C# и методов CIL.

Таблица 9.2. Соответствие имен операций C# и методов CIL

Внутренняя операция C#	Представление CIL
`--`	`op_Decrement()`
`++`	`op_Increment()`
`+`	`op_Addition()`
`-`	`op_Subtraction()`
`*`	`op_Multiply()`
`/`	`op_Division()`
`==`	`op_Equality()`
`>`	`op_GreaterThan()`
`<`	`op_LessThan()`
`!=`	`op_Inequality()`
`>=`	`op_GreaterThanOrEqual()`
`<=`	`op_LessThanOrEqual()`
`-=`	`op_SubtractionAssignment()`
`+=`	`op_AdditionAssignment()`

Использование перегруженных операций в языках, не поддерживающих перегрузку операций

Понимание того, как перегруженные операции представлены в программном коде CIL интересно не только с академической точки зрения. Чтобы осознать практическую пользу этих знаний, вспомните о том, что возможность перегрузки операций поддерживается *не всеми* языками, предназначенными для .NET. Как, например, добавить пару типов `Point` в программу, созданную на языке, не поддерживающем перегрузку операций?

Одним из подходов является создание "нормальных" открытых членов, которые будут решать ту же задачу, что и перегруженные операции. Например, можно добавить в `Point` методы `Add()` и `Subtract()`, которые будут выполнять работу, соответствующую операциям + и –.

```
// Экспозиция семантики перегруженных операций
// с помощью простых членов-функций.
public struct Point
{
...
    // Представление операции + с помощью Add()
    public static Point Add (Point p1, Point p2)
    { return p1 + p2; }

    // Представление операции - с помощью Subtract()
    public static Point Subtract (Point p1, Point p2)
    { return p1 - p2; }
}
```

С такими модификациями тип `Point` способен демонстрировать соответствующие функциональные возможности, используя любые подходы, предлагаемые в рамках данного языка. Пользователи C# могут применять операции + и – или же вызывать `Add()`/`Subtract()`.

```
// Использование операции + или Add().
Console.WriteLine("ptOne + ptTwo: {0} ", ptOne + ptTwo);
Console.WriteLine("Point.Add(ptOne, ptTwo): {0} ",
    Point.Add(ptOne, ptTwo));

// Использование операции - или Subtract().
Console.WriteLine("ptOne - ptTwo: {0} ", ptOne - ptTwo);
Console.WriteLine("Point.Subtract(ptOne, ptTwo): {0} ",
    Point.Subtract(ptOne, ptTwo));
```

Языки, в которых не допускается перегрузка операций, могут использовать только открытые статические методы. В качестве альтернативы можно предложить непосредственный вызов специальных именованных методов, создаваемых компилятором.

Рассмотрим исходный вариант языка программирования VB .NET. При построении консольного приложения VB .NET, ссылающегося на тип `Point`, вы можете добавлять или вычитать типы `Point`, используя "специальные CIL-имена", например:

```
' Предполагается, что данное приложение VB .NET
' имеет доступ к типу Point.
Module OverLoadedOpClient
   Sub Main()
      Dim p1 As Point
      p1.x = 200
      p1.y = 9

      Dim p2 As Point
      p2.x = 9
      p2.y = 983

      ' Не так красиво, как вызов AddPoints(),
      ' но зато работает.
      Dim bigPoint = Point.op_Addition(p1, p2)
      Console.WriteLine("Большая точка {0}", bigPoint)
   End Sub
End Module
```

Как видите, языки программирования .NET, не предусматривающие перегрузку операций, способны непосредственно вызывать внутренние методы CIL, как "обычные" методы. Такое решение нельзя назвать слишком "изящным", но оно работает.

Замечание. Текущая версия VB .NET (Visual Basic .NET 2005) перегрузку операций поддерживает. Однако для других (многочисленных) управляемых языков, не поддерживающих перегрузку операций, знание "специальных имен" соответствующих методов CIL может оказаться очень полезным.

Заключительные замечания о перегрузке операций

Вы могли убедиться в том, что C# обеспечивает возможность построения типов, по-своему отвечающих на встроенные всем известные операции. Перед тем как перейти к непосредственной модификации классов для поддержки такого поведения, вы должны убедиться в том, что для операций, которым вы хотите назначить перегрузку, такая перегрузка имеет смысл.

Предположим, например, что вы хотите использовать перегрузку операции умножения для класса `Engine` (мотор). Что тогда должно означать умножение двух объектов `Engine`? Не понятно. Перегрузка операций, в общем, оказывается полезной только тогда, когда строятся полезные типы. Строки, точки, прямоугольники и шестиугольники являются хорошими объектами для перегрузки операций. А люди, менеджеры, автомобили, наушники и бейсбольные кепки — нет. Если перегруженная операция делает более *трудным* понимание функциональных возможностей типа пользователем, то лучше перегрузку не использовать. Используйте указанную возможность с умом.

Исходный код. Проект OverloadedOps размещен в подкаталоге, соответствующем главе 9.

Пользовательские преобразования типов

Рассмотрим тему, близко связанную с перегрузкой операций: это пользовательские правила преобразования типов. В начале нашего рассмотрения мы кратко обсудим явные и неявные преобразования числовых данных и соответствующих типов класса.

Преобразования чисел

В случае встроенных числовых типов (`sbyte`, `int`, `float` и т.д.) *явное преобразование* требуется тогда, когда вы пытаетесь сохранить большее значение в меньшем контейнере, поскольку при этом может происходить потеря данных. По сути, это способ сказать компилятору примерно следующее: "Не беспокойся, я знаю, что делаю!" С другой стороны, *неявное преобразование* происходит автоматически, когда вы пытаетесь разместить в типе-адресате тип меньших размеров, в результате чего потери данных не происходит.

```
static void Main()
{
   int a = 123;
   long b = a;         // Неявное преобразование из int в long
   int c = (int) b;    // Явное преобразование из long в int
}
```

Преобразования типов класса

Как показано в главе 4, типы класса могут быть связаны классическим отношением наследования (отношение "is-a"). В этом случае в C# процесс преобразования позволяет сдвигаться вверх или вниз по иерархии классов. Например, производный класс всегда можно неявно преобразовать в базовый тип. Однако если вы захотите сохранить базовый тип класса в производной переменной, придется выполнить явное преобразование.

```
// Два связанных типа класса.
class Base{}
class Derived : Base{}

class Program
{
   static void Main()
   {
      // Неявное преобразование из производного в базовый.
      Base myBaseType;
      myBaseType = new Derived();

      // Для сохранения базовой ссылки в производном типе
      // следует выполнить явное преобразование.
      Derived myDerivedType = (Derived)myBaseType;
   }
}
```

Здесь явное преобразование работает благодаря тому, что классы `Base` и `Derived` связаны классическим отношением наследования. Но что делать в том случае, когда вы хотите связать преобразованием два типа класса, принадлежащие *разным иерархиям*? Если классы не связаны классическим наследованием, явное преобразование помочь ничем не сможет.

В соответствующем ключе рассмотрим типы, характеризуемые значениями. Предположим, что у нас есть две .NET-структуры с именами `Square` (квадрат) и `Rectangle` (прямоугольник). Поскольку структуры не могут использовать классическое наследование, нет и естественного способа взаимного преобразования этих явно связанных типов (в предположении о том, что такое преобразование имеет смысл).

Конечно, проблему можно решить с помощью создания в этих в структурах вспомогательных методов (например, `Rectangle.ToSquare()`), но в C# можно создавать пользовательские подпрограммы преобразования, позволяющие соответствующим типам по-своему отвечать на операцию `()`. Так, при правильной конфигурации типа `Square` вы получите возможность использовать следующий синтаксис для явного преобразования этих типов структуры.

```
// Превращение прямоугольника в квадрат.
Rectangle rect;
rect.Width = 3;
rect.Height = 10;
Square sq = (Square)rect;
```

Создание пользовательских подпрограмм преобразования

В C# есть два ключевых слова, `explicit` и `implicit`, предназначенные для управления тем, как типы должны отвечать на попытки преобразования. Предположим, что у нас есть следующие определения структур.

```
public struct Rectangle
{
    // Открыты для простоты,
    // но ничто не мешает инкапсулировать их в виде свойств.
    public int Width, Height;

    public void Draw()
    { Console.WriteLine("Отображение прямоугольника."); }

    public override string ToString()
    {
        return string.Format("[Ширина = {0}; Высота = {1}]",
            Width, Height);
    }
}

public struct Square
{
    public int Length;
```

```
    public void Draw()
    { Console.WriteLine("Отображение квадрата."); }

    public override string ToString()
    { return string.Format("[Сторона = {0}]", Length); }

    // Rectangle (прямоугольник) можно явно преобразовать
    // в Square (квадрат).
    public static explicit operator Square(Rectangle r)
    {
        Square s;
        s.Length = r.Width;
        return s;
    }
}
```

Обратите внимание на то, что на этот раз для типа `Rectangle` определяется операция явного преобразования. Как и при перегрузке встроенных операций, в C# для подпрограмм преобразования используется ключевое слово `operator` (в совокупности с ключевым словом `explicit` или `implicit`) и эти подпрограммы должны определяться, как статические. Входным параметром является объект, который вы хотите преобразовать, а возвращаемое значение — это объект, в который поступающий объект превращается.

```
public static explicit operator Square(Rectangle r)
{...}
```

Здесь предполагается, что квадрат (который является геометрической фигурой с равными сторонами) можно получить на основе ширины прямоугольника. Поэтому вы можете превратить `Rectangle` (прямоугольник) в `Square` (квадрат) так.

```
static void Main(string[] args)
{
    Console.WriteLine("***** Забавы с преобразованиями *****\n");

    // Создание прямоугольника 10 x 5.
    Rectangle rect;
    rect.Width = 10;
    rect.Height = 5;
    Console.WriteLine("rect = {0}", rect);
    // Преобразование прямоугольника в квадрат 10 x 10.
    Square sq = (Square)rect;
    Console.WriteLine("sq = {0}", sq);
    Console.ReadLine();
}
```

Наверное, от превращения прямоугольников в квадраты в рамках одного контекста не слишком много пользы, но предположим, что у нас есть функция, которая предполагает использование типов `Square`.

```
// Этот метод требует использования типа Square.
private static void DrawSquare(Square sq)
{
    sq.Draw();
}
```

Используя нашу операцию явного преобразования, мы можем передавать этой функции типы Square.

```
static void Main(string[] args)
{
...
    // Преобразование Rectangle в Square для вызова метода.
    DrawSquare((Square)rect);
}
```

Варианты явного преобразования для типа Square

Теперь вы можете явно превращать прямоугольники в квадраты, но рассмотрим еще несколько вариантов явного преобразования. Поскольку у квадрата стороны равны, можно явно преобразовать System.Int32 в Square (длина стороны квадрата будет равна значению поступающего целого числа). Аналогично можно изменить определение Square, если требуется обеспечить преобразование из Square в System.Int32. Вот логика соответствующего вызова.

```
static void Main(string[] args)
{
...
    // Преобразование System.Int32 в Square.
    Square sq2 = (Square)90;
    Console.WriteLine("sq2 = {0}", sq2);

    // Преобразование Square в System.Int32.
    int side = (int)sq2;
    Console.WriteLine("Длина стороны sq2 = {0}", side);
}
```

А вот как следует обновить определение типа Square.

```
public struct Square
{
...
    public static explicit operator Square(int sideLength)
    {
        Square newSq;
        newSq.Length = sideLength;
        return newSq;
    }

    public static explicit operator int (Square s)
    {return s.Length;}
}
```

Выглядит немного странно, не так ли? Честно говоря, преобразование из Square в System.Int32 не является интуитивно очевидной (или полезной) операцией. Однако она демонстрирует одну очень важную особенность пользовательских подпрограмм преобразования: компилятору “все равно” из чего и во что вы преобразуете — важно, чтобы ваш программный код был синтаксически правильным. Так что, как в случае с перегрузкой операций, только из того, что вы можете

создать операцию явного преобразования для данного типа, совсем не следует, что вы обязаны это делать. Как правило, этот подход оказывается наиболее полезным тогда, когда создаются типы структуры .NET, поскольку такие типы не могут использовать иерархии классического наследования (для которых соответствующие преобразования реализуются автоматически).

Определение подпрограмм неявного преобразования

До этого момента мы с вами создавали пользовательские операции явного преобразования. Но что можно сказать о следующем *неявном* преобразовании?

```
static void Main(string[] args)
{
...
   // Попытка выполнить неявное преобразование?
   Square s3;
   s3.Length = 83;
   Rectangle rect2 = s3;
}
```

Как вы можете догадаться сами, этот программный код скомпилирован не будет, поскольку в нем не предлагается никакой подпрограммы неявного преобразования для типа `Rectangle`. Тут нас подстерегает "ловушка": в одном и том же типе нельзя определять явные и неявные функции преобразования, не отличающиеся по типу возвращаемого значения или по набору параметров. Может показаться, что это правило является слишком ограничивающим, но не следует забывать о том, что даже если тип определяет подпрограмму *неявного* преобразования, вызывающая сторона "имеет право" использовать синтаксис *явного* преобразования!

Запутались? Чтобы прояснить ситуацию, добавим в структуру `Rectangle` подпрограмму неявного преобразования, используя ключевое слово C# `implicit` (в следующем программном коде предполагается, что ширина результирующего `Rectangle` получается с помощью умножения стороны `Square` на 2).

```
public struct Rectangle
{
...
   public static implicit operator Rectangle(Square s)
   {
      Rectangle r;
      r.Height = s.Length;

      // Ширина нового прямоугольника равна
      // удвоенной длине стороны квадрата.
      r.Width = s.Length * 2;
      return r;
   }
}
```

С такими изменениями вы получаете возможность преобразовывать указанные типы так.

```
static void Main(string[] args)
{
...
    // Неявное преобразование: все OK!
    Square s3;
    s3.Length= 83;
    Rectangle rect2 = s3;
    Console.WriteLine("rect2 = {0}", rect2);
    DrawSquare(s3);

    // Синтаксис явного преобразования: тоже OK!
    Square s4;
    s4.Length = 3;
    Rectangle rect3 = (Rectangle)s4;
    Console.WriteLine("rect3 = {0}", rect3);
...
}
```

Снова подчеркнем, что допускается определение подпрограмм и явного, и неявного преобразования для одного и того же типа, но только если отличаются их сигнатуры. Поэтому мы можем обновить `Square` так, как показано ниже.

```
public struct Square
{
...
    // Можно вызывать как
    // Square sq2 = (Square)90;
    // или как
    // Square sq2 = 90;
    public static implicit operator Square(int sideLength)
    {
        Square newSq;
        newSq.Length = sideLength;
        return newSq;

        // Должно вызываться как
        // int side = (Square)mySquare;
        public static explicit operator int (Square s)
        { return s.Length; }
    }
}
```

Внутреннее представление пользовательских подпрограмм преобразования

Как и в случае перегруженных операций, те методы, которые обозначены ключевыми словами `implicit` или `explicit`, получают "специальные имена" в терминах CIL: `op_Implicit` и `op_Explicit` соответственно (рис. 9.2).

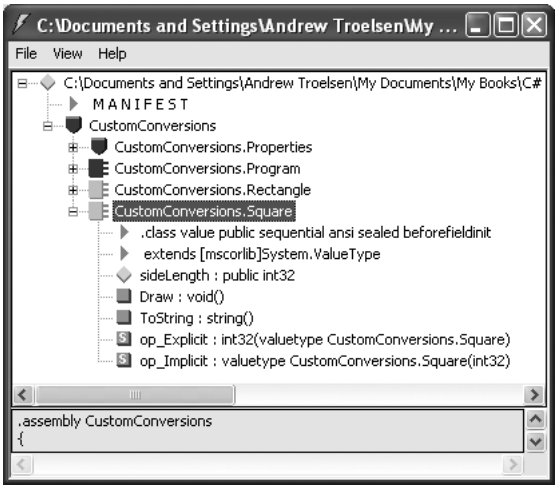

Рис. 9.2. Представление пользовательских подпрограмм преобразования в терминах CIL

На этом мы завершаем обзор возможностей пользовательских подпрограмм преобразования. Как и в случае перегруженных операций, соответствующий синтаксис является лишь сокращенным вариантом определения "нормальных" членов-функций, и с этой точки зрения он не является обязательным.

Исходный код. Проект CustomConversions размещен в подкаталоге, соответствующем главе 9.

Ключевые слова C#, предназначенные для более сложных конструкций

В завершение главы мы рассмотрим ряд ключевых слов C#, применение которых требует от разработчика несколько большего опыта в программировании:

- `checked`/`unchecked`;
- `unsafe`/`stackalloc`/`fixed`/`sizeof`.

Сначала мы выясним, как с помощью ключевых слов `checked` и `unchecked` в C# обеспечивается автоматическое выявление условий переполнения и потери значимости при выполнении арифметических операций.

Ключевое слово checked

Вы, несомненно, прекрасно знаете, что любой числовой тип данных имеет свои строго заданные верхний и нижний пределы (значения которых можно выяснить программными средствами с помощью свойств `MaxValue` и `MinValue`). При выполнении арифметических операций с конкретным типом вполне возможно случайное *переполнение* блока хранения данного типа (попытка присвоения типу значения, которое оказывается больше максимально допустимого) или *потеря значимости* (попытка присвоения значения, которое оказывается меньше минимально допу-

стимого). Чтобы "идти в ногу" с CLR, обе эти возможности будут обозначаться, как "переполнение". (И переполнение, и потеря значимости приводят к созданию типа `System.OverflowException`. Типа `System.UnderflowException` в библиотеках базовых классов нет.)

Для примера предположим, что мы создали два экземпляра типа `System.Byte` (тип `byte` в C#), присвоив им значения, не превышающие максимального (255). При сложении значений этих типов (с условием преобразования результата в тип `byte`) хотелось бы предполагать, что результат будет точной суммой соответствующих членов.

```
namespace CheckedUnchecked
{
   class Program
   {
      static void Main(string[] args)
      {
         // Переполнение для System.Byte.
         Console.WriteLine("Макс. значение для byte равно {0}.",
            byte.MaxValue);
         Console.WriteLine("Мин. значение для byte равно {0}.",
            byte.MinValue);
         byte b1 = 100;
         byte b2 = 250;
         byte sum = (byte)(b1 + b2);

         // Значением sum должно быть 350, но...
         Console.WriteLine("sum = {0}", sum);
         Console.ReadLine();
      }
   }
}
```

Вывод этого приложения покажет, что `sum` содержит значение 94 (а не ожидаемое 350). Причина очень проста. Поскольку `System.Byte` может содержать только значения, находящиеся между 0 и 255 (что в итоге составляет 256 значений), `sum` будет содержать значение переполнения (350 – 256 = 94). Как видите, в отсутствие специальной коррекции переполнение происходит без генерирования исключений. Иногда скрытое переполнение не создает никаких проблем. В других случаях соответствующая потеря данных может быть совершенно неприемлемой.

Для обработки переполнений или потери значимости в приложении имеются две возможности. Первой возможностью является использование программистского опыта и квалификации с тем, чтобы обработать все условия переполнения вручную. Предполагая, что вы можете найти все условия переполнения в программе, можно было бы решить проблему, связанную с переполнением в предыдущем программном коде, как показано ниже.

```
// Использование int для sum, чтобы не допустить переполнения.
byte b1 = 100;
byte b2 = 250;
int sum = b1 + b2;
```

Конечно, проблемой этого подхода является то, что вы — человек, а значит, при всех ваших усилиях, могут остаться ошибки, ускользнувшие от вашего взгляда. Поэтому в C# предлагается ключевое слово `checked`. При помещении оператора (или блока операторов) в рамки контекста ключевого слова `checked` компилятор C# генерирует специальные CIL-инструкции, с помощью которых проверяются условия переполнения, возможные при выполнении сложения, умножения, вычитания или деления числовых типов данных. Если происходит переполнение, среда выполнения генерирует тип `System.OverflowException`. Рассмотрите следующую модификацию программы.

```
class Program
{
    static void Main(string[] args)
    {
        // Переполнение для System.Byte.
        Console.WriteLine("Макс. значение для byte равно {0}.",
            byte.MaxValue);
        byte b1 = 100;
        byte b2 = 250;

        try
        {
            byte sum = checked((byte)(b1 + b2));
            Console.WriteLine("sum = {0}", sum);
        }
        catch(OverflowException e)
        { Console.WriteLine(e.Message); }
    }
}
```

Здесь оператор сложения `b1` и `b2` помещается в контекст ключевого слова `checked`. Если вы хотите, чтобы проверка переполнения происходила для блока программного кода, можно взаимодействовать с ключевым словом `checked` так, как показано ниже.

```
try
{
    checked
    {
        byte sum = (byte)(b1 + b2);
        Console.WriteLine("sum = {0}", sum);
    }
}
catch(OverflowException e)
{
    Console.WriteLine(e.Message);
}
```

В любом случае соответствующий программный код будет проверяться на возможное переполнение автоматически, и если переполнение будет обнаружено, то будет сгенерировано соответствующее переполнению исключение.

Проверки переполнения для всего проекта

Если вы создаете приложение, которое не должно позволять скрытое переполнение ни при каких условиях, будет слишком утомительно указывать ключевое слово `checked` для каждой строки программного кода. В качестве альтернативы компилятор C# предлагает использовать флаг `/checked`. Когда этот флаг активизирован, *все* арифметические операции будут проверяться на переполнение без указания ключевого слова `checked`. Если обнаружится переполнение, вы получите `OverflowException` среды выполнения.

Чтобы активизировать этот флаг в Visual Studio 2005, откройте страницу свойств проекта и щелкните на кнопке Advanced на вкладке Build. В появившемся диалоговом окне отметьте флажок Check for arithmetic overflow/underflow (Проверять условия переполнения/потери значимости для арифметических операций), рис. 9.3.

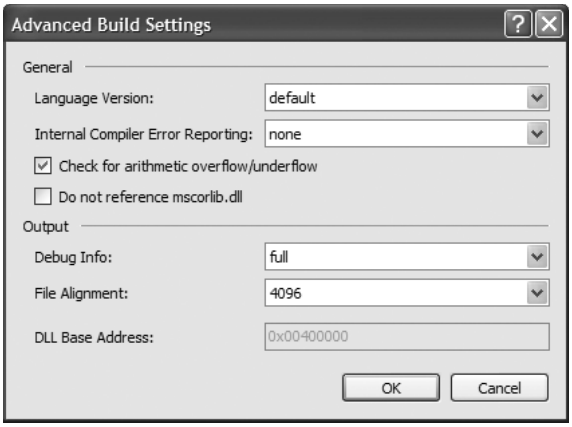

Рис. 9.3. Активизация проверки переполнения в Visual Studio 2005

Ясно, что эта установка оказывается очень полезной при отладке. После того как все связанные с переполнением исключения будут из программного кода удалены, флаг `/checked` для последующей компоновки можно отключить.

Ключевое слово unchecked

В предположении, что вы активизировали проверку переполнения для всего проекта, как разрешить игнорирование переполнений для тех блоков программного кода, где "молчаливая реакция" на переполнение вполне приемлема? Поскольку флаг `/checked` предполагает проверку *всей* арифметической логики, в языке C# предлагается ключевое слово `unchecked`, которое позволяет отключить генерирование `System.OverflowException` для конкретных случаев. Правила использования этого ключевого слова аналогичны правилам использования ключевого слова `checked`, и вы можете указать для него один оператор или блок операторов, например:

```
// Даже если флаг /checked активизирован,
// этот блок не генерирует исключения в среде выполнения.
unchecked
```

```
{
    byte sum = (byte)(b1 + b2);
    Console.WriteLine("sum = {0}", sum);
}
```

Подводя итоги обсуждения ключевых слов C# `checked` и `unchecked`, снова подчеркнем, что по умолчанию среда выполнения .NET игнорирует условия переполнения, возникающие при использовании арифметических операций. Если вы хотите селективно контролировать условия переполнения для отдельных операторов, используйте ключевое слово `checked`. Если нужно контролировать ошибки переполнения во всем приложении, укажите флаг `/checked`. Наконец, можно использовать ключевое слово `unchecked`, если у вас есть блок программного кода, для которого переполнение приемлемо (и поэтому оно не должно генерировать исключение в среде выполнения).

Исходный код. Проект CheckedUnchecked размещен в подкаталоге, соответствующем главе 9.

Работа с типами указателя

Из главы 3 вы узнали, что платформа .NET определяет две главные категории данных: типы, характеризуемые значениями, и типы, характеризуемые ссылками (ссылочные типы). Однако, справедливости ради, следует сказать, что имеется и третья категория: это *типы указателя*. Для работы с типами указателя предлагаются специальные операции и ключевые слова, с помощью которых можно "обойти" схему управления памятью CLR и "взять управление в свои руки" (табл. 9.3).

Таблица 9.3. Операции и ключевые слова C# для работы с указателями

Операция или ключевое слово	Описание
`*`	Используется для создания *переменной указателя* (т.е. переменной, представляющей непосредственно адресуемую точку в памяти). Как и в C(++), тот же знак используется для операции разыменования указателя (т.е. для операции, которая возвратит значение, размещенное по адресу, указанному операндом)
`&`	Используется для получения адреса переменной в памяти
`->`	Используется для доступа к полям типа, представленным указателем (небезопасная версия операции, обозначаемой в C# точкой)
`[]`	Операция `[]` (в небезопасном контексте) позволяет индексировать элемент, на который указывает переменная указателя. (Обратите внимание на аналогию между переменной указателя и операцией `[]` в C(++).)
`++`, `--`	В небезопасном контексте к типам указателя могут применяться операции приращения и отрицательного приращения
`+`, `-`	В небезопасном контексте к типам указателя могут применяться операции сложения и вычитания
`==`, `!=`, `<`, `>`, `<=`, `=>`	В небезопасном контексте к типам указателя могут применяться операции сравнения и проверки на тождественность
`stackalloc`	В небезопасном контексте можно использовать ключевое слово `stackalloc`, чтобы размещать массивы C# в стеке
`fixed`	В небезопасном контексте можно использовать ключевое слово `fixed`, временно фиксирующее переменную с тем, чтобы можно было найти ее адрес

Перед рассмотрением деталей позвольте заметить, что необходимость в использовании типов указателя возникает *очень редко*, если она возникает вообще. Хотя C# и позволяет "спуститься" на уровень манипуляций с указателями, следует понимать, что среда выполнения .NET не имеет никакого представления о ваших намерениях. Поэтому если вы ошибетесь в направлении указателя, то за последствия будете отвечать сами. Если учитывать это, то когда же на самом деле возникает необходимость использования типов указателя? Есть две стандартные ситуации.

- Вы стремитесь оптимизировать работу определенных частей своего приложения путем непосредственной манипуляции памятью вне пределов управления CLR.
- Вы хотите использовать методы C-библиотеки `*.dll` или COM-сервера, требующие ввода указателей в виде параметров.

Если вы решите использовать указанную возможность языка C#, необходимо информировать `csc.exe` об этих намерениях, указав разрешение для проекта поддерживать "небезопасный программный код". Чтобы сделать это с командной строки компилятора C# (`csc.exe`), просто укажите в качестве аргумента флаг `/unsafe`. В Visual Studio 2005 вы должны перейти на страницу свойств проекта и активизировать опцию Allow Unsafe Code (Разрешать использование небезопасного программного кода) на вкладке Build (рис. 9.4).

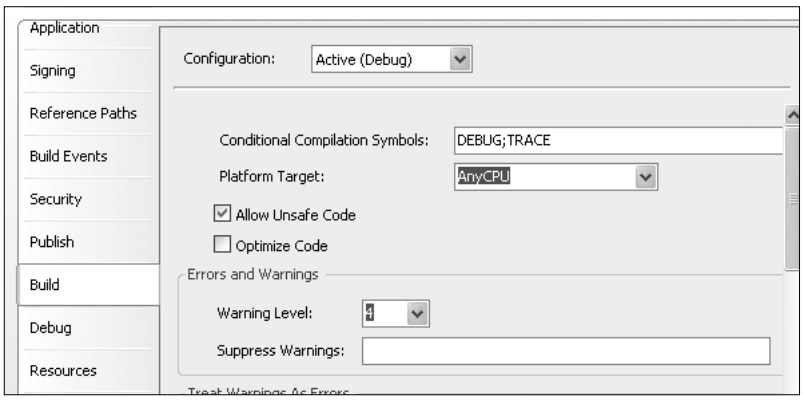

Рис. 9.4. Разрешение небезопасного программного кода Visual Studio 2005

Ключевое слово unsafe

В следующих примерах предполагается, что вы имеете опыт работы с указателями в C(++). Если это не так, не слишком отчаивайтесь. Подчеркнем еще раз, что создание небезопасного программного кода не является типичной задачей в большинстве приложений .NET. Если вы хотите работать с указателями в C#, то должны специально объявить блок программного кода "небезопасным", используя для этого ключевое слово `unsafe` (как вы можете догадаться сами, весь программный код, не обозначенный ключевым `unsafe`, автоматически считается "безопасным").

```
unsafe
{
   // Операторы для работы с указателями.
}
```

Кроме объявления контекста небезопасного программного кода, вы можете строить "небезопасные" структуры, классы, члены типов и параметры. Вот несколько примеров, на которые следует обратить внимание.

```
// Вся эта структура является 'небезопасной'
// и может использоваться только в небезопасном контексте.
public unsafe struct Node
{
   public int Value;
   public Node* Left;
   public Node* Right;
}

// Эта структура является безопасной, но члены Node* - нет.
// Строго говоря, получить доступ к 'Value' извне небезопасного
// контекста можно, а к 'Left' и 'Right' - нет.
public struct Node
{
   public int Value;

   // К этим элементам можно получить доступ только
   // в небезопасном контексте!
   public unsafe Node* Left;
   public unsafe Node* Right;
}
```

Методы (как статические, так и уровня экземпляра) тоже можно обозначать, как небезопасные. Предположим, например, вы знаете, что некоторый статический метод использует логику указателей. Чтобы гарантировать вызов данного метода только в небезопасном контексте, можно определить метод так, как показано ниже.

```
unsafe public static void SomeUnsafeCode()
{
   // Операторы для работы с указателями.
}
```

В такой конфигурации требуется, чтобы вызывающая сторона обращалась к `SomeUnsafeCode()` так.

```
static void Main(string[] args)
{
   unsafe
   {
      SomeUnsafeCode();
   }
}
```

Если же не обязательно, чтобы вызывающая сторона делала вызов в небезопасном контексте, то можно не указывать ключевое слово `unsafe` в методе `SomeUnsafeCode()` и записать следующее:

```
public static void SomeUnsafeCode()
{
    unsafe
    {
        // Операторы для работы с указателями.
    }
}
```

что должно упростить вызов:

```
static void Main(string[] args)
{
    SomeUnsafeCode();
}
```

Работа с операциями * и &

После создания небезопасного контекста вы можете строить указатели на типы с помощью операции `*` и получать адреса заданных указателей с помощью операции `&`. В C# операция `*` применяется только к соответствующему типу, а не как префикс ко всем именам переменных указателя. Например, в следующем фрагменте программного кода объявляются две переменные типа `int*` (указатель на целое).

```
// Нет! В C# это некорректно!
int *pi, *pj;

// Да! Это в C# правильно.
int* pi, pj;
```

Рассмотрим следующий пример.

```
unsafe
{
    int myInt;

    // Определения указателя типа int
    // и присваивание ему адреса myInt.
    int* ptrToMyInt = &myInt;

    // Присваивание значения myInt
    // с помощью разыменования указателя.
    *ptrToMyInt = 123;

    // Печать статистики.
    Console.WriteLine("Значение myInt {0}", myInt);
    Console.WriteLine("Адрес myInt {0:X}", (int)&ptrToMyInt);
}
```

Небезопасная (и безопасная) функция Swap

Конечно, объявление указателей на локальные переменные с помощью простого присваивания им значений (как в предыдущем примере) никогда не требуется. Чтобы привести более полезный пример небезопасного программного кода, предположим, что вы хотите создать функцию обмена, используя арифметику указателей.

```
unsafe public static void UnsafeSwap(int* i, int* j)
{
    int temp = *i;
    *i = *j;
    *j = temp;
}
```

Очень похоже на C, не так ли? Однако с учетом знаний, полученных из главы 3, вы должны знать, что можно записать следующую безопасную версию алгоритма обмена, используя ключевое слово C# `ref`.

```
public static void SafeSwap(ref int i, ref int j)
{
    int temp = i;
    i = j;
    j = temp;
}
```

Функциональные возможности каждого из этих методов идентичны, и это еще раз подтверждает, что работа напрямую с указателями в C# требуется редко. Ниже показана логика вызова.

```
static void Main(string[] args)
{
    Console.WriteLine("*** Вызов метода с небезопасным кодом ***");

    // Значения для обмена.
    int i = 10,
    j = 20;

    // 'Безопасный' обмен значениями.
    Console.WriteLine("\n***** Безопасный обмен *****");
    Console.WriteLine("Значения до обмена: i = {0}, j = {1}", i, j);
    SafeSwap(ref i, ref j);
    Console.WriteLine("Значения после обмена: i = {0}, j = {1}", i, j);

    // 'Небезопасный' обмен значениями.
    Console.WriteLine("\n***** Небезопасный обмен *****");
    Console.WriteLine("Значения до обмена: i = {0}, j = {1}", i, j);
    unsafe { UnsafeSwap(&i, &j); }
    Console.WriteLine("Значения после обмена: i = {0}, j = {1}", i, j);
    Console.ReadLine();
}
```

Доступ к полям через указатели (операция –>)

Теперь предположим, что у нас определена структура `Point` и мы хотим объявить указатель на тип `Point`. Как и в C(++), для вызова методов или получения доступа к полям типа указателя необходимо использовать операцию доступа к полю указателя (`->`). Как уже упоминалось в табл. 9.3, это небезопасная версия стандартной (безопасной) операции, обозначаемой точкой (`.`). Фактически, используя операцию разыменования указателя (`*`), можно снять косвенность указателя, чтобы (снова) вернуться к применению нотации, обозначаемой точкой. Рассмотрите следующий программный код.

```
struct Point
{
   public int x;
   public int y;
   public override string ToString()
   { return string.Format("({0}, {1})", x, y); }
}
static void Main(string[] args)
{
   // Доступ к членам через указатели.
   unsafe
   {
      Point point;
      Point* p = &point;
      p->x = 100;
      p->y = 200;
      Console.WriteLine(p->ToString());
   }
   // Доступ к членам через разыменовывание указателей.
   unsafe
   {
      Point point;
      Point* p = &point;
      (*p).x = 100;
      (*p).y = 200;
      Console.WriteLine( (*p).ToString());
   }
}
```

Ключевое слово stackalloc

В небезопасном контексте может понадобиться объявление локальной переменной, размещаемой непосредственно в памяти стека вызовов (и таким образом не подлежащей "утилизации" при сборке мусора .NET). Чтобы сделать такое объявление, в C# предлагается ключевое слово `stackalloc`, являющееся C#-эквивалентом функции `_alloca` из библиотеки времени выполнения C. Вот простой пример.

```
unsafe
{
   char* p = stackalloc char[256];
   for (int k = 0; k < 256; k++)
      p[k] = (char)k;
}
```

Фиксация типа с помощью ключевого слова fixed

Как показывает предыдущий пример, задачу размещения элемента в памяти в рамках небезопасного контекста можно упростить с помощью ключевого слова `stackalloc`. В силу самой природы этой операции соответствующая память очищается, как только происходит возврат из метода размещения (поскольку память выбирается из стека). Но рассмотрим более сложный пример. В процессе обсуждения операции `->` вы создали характеризуемый значением тип `Point`. Подобно всем типам, характеризуемым значениями, выделенная для него память в стеке освобождается сразу же после исчезновения контекста выполнения. Теперь, для примера, предположим, что тип `Point` был определен как *ссылочный* тип.

```
class Point    // <= Теперь это класс!
{
   public int x;
   public int y;
   public override string ToString()
   { return string.Format("({0}, {1})", x, y); }
}
```

Вы хорошо знаете о том, что если вызывающая сторона объявляет переменную типа `Point`, для нее выделяется динамическая память, являющаяся объектом для сборки мусора. Тогда возникает вопрос: что произойдет, если небезопасный контекст попытается взаимодействовать с соответствующим объектом (или любым другим объектом в динамической памяти)? Поскольку сборка мусора может начаться в любой момент, представьте себе всю болезненность доступа к членам `Point`, когда идет очистка динамической памяти. Теоретически возможно, что небезопасный контекст будет пытаться взаимодействовать с членом, который больше не доступен или занимает новое положение в динамической памяти, "пережив" очередную генерацию чистки (и это, очевидно, является проблемой).

Чтобы блокировать переменную ссылочного типа в памяти из небезопасного контекста, в C# предлагается ключевое слово `fixed`. Оператор `fixed` устанавливает указатель на управляемый тип и "закрепляет" переменную на время выполнения оператора. Без ключевого слова `fixed` в применении указателей на управляемые переменные было бы мало смысла, поскольку в результате сборки мусора такие переменные могут перемещаться непредсказуемым образом. (На самом деле компилятор C# вообще не позволит установить указатель на управляемую переменную, если в операторе не используется ключевое слово `fixed`.)

Так что если вы создадите тип `Point` (сейчас переопределенный, как класс) и захотите взаимодействовать с его членами, то должны записать следующий программный код (иначе возникнет ошибка компиляции).

```
unsafe public static void Main()
{
   Point pt = new Point();
   pt.x = 5;
   pt.y = 6;

   // Фиксация pt, чтобы не допустить перемещения
   // или удаления при сборке мусора.
   fixed (int* p = &pt.x)
```

```
    {
    // Переменная int* используется здесь.
    }
    // Теперь pt не зафиксирована и может быть убрана
    // сборщиком мусора.
    Console.WriteLine ("Значение Point: {0}", pt);
}
```

В сущности, ключевое слово `fixed` позволяет строить операторы, закрепляющие ссылочную переменную в памяти, чтобы ее адрес оставался постоянным на время выполнения оператора. Для гарантии безопасности обязательно фиксируйте ссылки при взаимодействии со ссылочными типами из небезопасного контекста программного кода.

Ключевое слово sizeof

В заключение обсуждения вопросов, связанных с небезопасным контекстом в C#, рассмотрим ключевое слово `sizeof`. Как и в C(++), ключевое слово C# `sizeof` используется для того, чтобы выяснить размер в байтах типа, характеризуемого значениями (но не ссылочного типа), и это ключевое слово может использоваться только в рамках небезопасного контекста. Очевидно, что указанная возможность может оказаться полезной при взаимодействии с неуправляемыми API, созданными на базе C. Использовать ее очень просто.

```
unsafe
{
   Console.WriteLine("Длина short равна {0}.", sizeof(short));
   Console.WriteLine("Длина int равна {0}.", sizeof(int));
   Console.WriteLine("Длина long равна {0}.", sizeof(long));
}
```

Поскольку `sizeof` может оценить число байтов для любого элемента, производного от `System.ValueType`, можно получать размеры пользовательских структур. Допустим, мы определили следующую структуру.

```
struct MyValueType
{
   public short s;
   public int i;
   public long l;
}
```

Тогда ее размеры можно выяснить так.

```
unsafe
{
   Console.WriteLine("Длина short равна {0}.", sizeof(short));
   Console.WriteLine("Длина int равна {0}.", sizeof(int));
   Console.WriteLine("Длина long равна {0}.", sizeof(long));
   Console.WriteLine("Длина MyValueType равна {0}.",
      sizeof(MyValueType));
}
```

Исходный код. Проект UnsafeCode размещен в подкаталоге, соответствующем главе 9.

Директивы препроцессора C#

Подобно многим другим языкам из семейства C, в C# поддерживаются различные символы, позволяющие влиять на процесс компиляции. Перед рассмотрением директив препроцессора C# согласуем соответствующую терминологию. Термин "директива препроцессора C#" не вполне точен. Фактически этот термин используется только для согласованности с языками программирования C и C++. В C# нет отдельного шага препроцессора. Директивы препроцессора в C# являются составной частью процесса лексического анализа компилятора.

Так или иначе, синтаксис директив препроцессора C# очень похож на синтаксис соответствующих директив остальных членов семейства C в том, что эти директивы всегда имеют префикс, обозначенный знаком "диез" (#). В табл. 9.4 описаны некоторые из наиболее часто используемых директив (подробности можно найти в документации .NET Framework 2.0 SDK).

Таблица 9.4. Типичные директивы препроцессора C#

Директивы	Описание
`#region`, `#endregion`	Используются для обозначения разделов стягиваемого исходного кода
`#define`, `#undef`	Используются для определения и отмены определения символов условной компиляции
`#if`, `#elif`, `#else`, `#endif`	Используются для условного пропуска разделов исходного кода (на основе указанных символов компиляции)

Разделы программного кода

Возможно, одной из самых полезных директив препроцессора являются `#region` и `#endregion`. Используя эти признаки, вы указываете блок программного кода, который можно скрыть от просмотра и идентифицировать информирующим текстовым маркером. Использование разделов программного кода может упростить обслуживание больших файлов `*.cs`. Можно, например, создать один раздел для конструкторов типа, другой — для свойств и т.д.

```
class Car
{
    private string petName;
    private int currSp;
    #region Constructors
    public Car()
    { ... }
    public Car Car(int currSp, string petName)
    { ... }
    #endregion
    #region Properties
    public int Speed
    { ... }
    public string Name
    { ... }
    #endregion
}
```

При помещении указателя мыши на маркер свернутого раздела вы получите снимок программного кода, спрятанного за соответствующим названием (рис. 9.5).

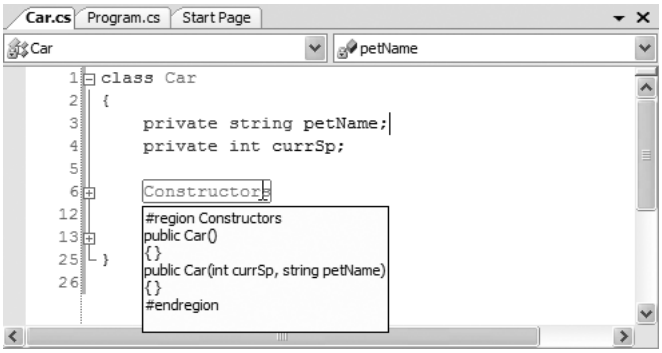

Рис. 9.5. Разделы программного кода за работой

Условная компиляция

Другой пакет директив препроцессора (`#if`, `#elif`, `#else`, `#endif`) позволяет выполнить компиляцию блока программного кода по условию, базируясь на предварительно заданных символах. Классическим вариантом использования этих директив является идентификация блока программного кода, который компилируется только при отладке (а не при окончательной компоновке).

```
class Program
{
   static void Main(string[] args)
   {
      // Этот программный код выполняется только при отладочной
      // компиляции проекта.
      #if DEBUG
      Console.WriteLine("Каталог приложения: {0}",
         Environment.CurrentDirectory);
      Console.WriteLine("Блок: {0}",
         Environment.MachineName);
      Console.WriteLine("ОС: {0}",
         Environment.OSVersion);
      Console.WriteLine("Версия .NET: {0}",
         Environment.Version);
      #endif
   }
}
```

Здесь выполняется проверка на символ `DEBUG`. Если он присутствует, выводится ряд данных состояния, для чего используются соответствующие статические члены класса `System.Environment`. Если символ `DEBUG` не обнаружен, то программный код, размещенный между `#if` и `#endif`, компилироваться не будет и в результирующий компоновочный блок не войдет, т.е. будет фактически проигнорирован.

По умолчанию Visual Studio 2005 всегда определяет символ DEBUG, однако такое поведение можно отменить путем снятия отметки флажка Define DEBUG constant (Определить константу DEBUG) на вкладке Build (Сборка), размещенной на странице Properties (Свойства) вашего проекта. В предположении о том, что этот обычно генерируемый символ DEBUG отключен, можно определить этот символ для каждого файла в отдельности, используя директиву препроцессора #define.

```
#define DEBUG
using System;

namespace Preprocessor
{
    class ProcessMe
    {
        static void Main(string[] args)
        {
            // Программный код, подобный показанному выше...
        }
    }
}
```

Замечание. Директивы #define в файле с программным кодом C# должны быть указаны до всех остальных.

Можно также определять свои собственные символы препроцессора. Предположим, например, что у нас есть класс C#, который должен компилироваться немного иначе в рамках дистрибутива Mono .NET (см. главу 1). Используя #define, можно определить символ MONO_BUILD для каждого файла.

```
#define DEBUG
#define MONO_BUILD

using System;

namespace Preprocessor
{
    class Program
    {
        static void Main(string[] args)
        {
            #if MONO_BUILD
                Console.WriteLine("Компиляция для Mono!");
            #else
                Console.WriteLine("Компиляция для Microsoft .NET");
            #endif
        }
    }
}
```

Чтобы создать символ, применимый для всего проекта, используйте текстовый блок Conditional compilation symbols (Символы условной компиляции), размещенный на вкладке Build (Сборка) страницы свойств проекта (рис. 9.6).

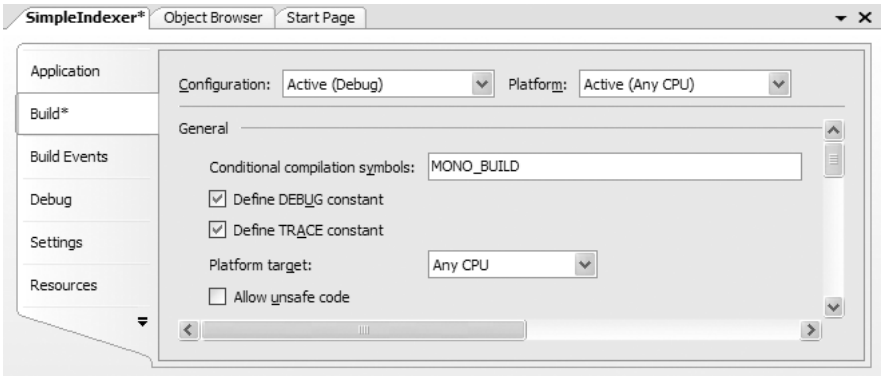

Рис. 9.6. Определение символа препроцессора для применения в рамках всего проекта

Резюме

Целью этой главы является более глубокое изучение возможностей языка программирования C#. Глава началась с обсуждения ряда достаточно сложных конструкций программирования (методов индексатора, перегруженных операций и пользовательских подпрограмм преобразования). Затем был рассмотрен небольшой набор не слишком широко известных ключевых слов (таких, как `sizeof`, `checked`, `unsafe` и т.д.), обсуждение которых естественно привело к рассмотрению вопросов непосредственной работы с типами указателя. При исследовании типов указателя было показано, что в подавляющем большинстве приложений C# для использования типов указателя нет *никакой* необходимости.