Структуры

Как вам должно быть уже известно, классы относятся к ссылочным типам данных. Это означает, что объекты конкретного класса доступны по ссылке, в отличие от значений простых типов, доступных непосредственно. Но иногда прямой доступ к объектам как к значениям простых типов оказывается полезно иметь, например, ради повышения эффективности программы. Ведь каждый доступ к объектам (даже самым мелким) по ссылке связан с дополнительными издержками на расход вычислительных ресурсов и оперативной памяти. Для разрешения подобных затруднений в C# предусмотрена структура, которая подобна классу, но относится к типу значения, а не к ссылочному типу данных.

Структуры объявляются с помощью ключевого слова `struct` и с точки зрения синтаксиса подобны классам. Ниже приведена общая форма объявления структуры:

```
struct имя : интерфейсы
{
  // объявления членов
}
```

где *имя* обозначает конкретное имя структуры.

Одни структуры не могут наследовать другие структуры и классы или служить в качестве базовых для других структур и классов. (Разумеется, структуры, как и все остальные типы данных в C#, наследуют класс `object`.) Тем не менее в структуре можно реализовать один или несколько интерфейсов, которые указываются после имени структуры списком через запятую. Как и у классов, у каждой структуры имеются свои члены: методы, поля, индексаторы, свойства, операторные методы и события. В структурах допускается также определять конструкторы, но не деструкторы. В то же время для структуры нельзя определить конструктор, используемый по умолчанию (т.е. конструктор без параметров). Дело в том, что конструктор, вызываемый по умолчанию, определяется для всех структур автоматически и не подлежит изменению. Такой конструктор инициализирует поля структуры значениями, задаваемыми по умолчанию. А поскольку структуры не поддерживают наследование, то их члены нельзя указывать как `abstract`, `virtual` или `protected`.

Объект структуры может быть создан с помощью оператора `new` таким же образом, как и объект класса, но в этом нет особой необходимости. Ведь когда используется оператор new, то вызывается конструктор, используемый по умолчанию. А когда этот оператор не используется, объект по-прежнему создается, хотя и не инициализируется. В этом случае инициализацию любых членов структуры придется выполнить вручную.

В приведенном ниже примере программы демонстрируется применение структуры для хранения информации о книге.

Листинг 1

```
// Продемонстрировать применение структуры.

using System;

// Определить структуру.
struct Book
{
  public string Author;
  public string Title;
  public int Copyright;

  public Book(string a, string t, int c) {
    Author = a;
    Title = t;
```

```
    Copyright = c;
  }
}

// Продемонстрировать применение структуры Book.
class StructDemo
{
  static void Main()
  {
    Book book1 = new Book("Герберт Шилдт",
                          "Полное руководство по C# 4.0",
                          2010);   // вызов явно заданного
конструктора

    Book book2 = new Book(); // вызов конструктора по умолчанию
    Book book3; // конструктор не вызывается

    Console.WriteLine(book1.Title + " by " + book1.Author +
                      ", (c) " + book1.Copyright);
    Console.WriteLine();

    if(book2.Title == null)
      Console.WriteLine("Член book2.Title пуст.");

    // А теперь ввести информацию в структуру book2.
    book2.Title = "О дивный новый мир";
    book2.Author = "Олдос Хаксли";
    book2.Copyright = 1932;
    Console.Write("Структура book2 теперь содержит: ");
    Console.WriteLine(book2.Title + " by " + book2.Author +
                      ", (c) " + book2.Copyright);

    Console.WriteLine();

//   Console.WriteLine(book3.Title);   // ошибка,   этот   член
структуры
                                    // надо сначала инициализировать
    book3.Title = "Красный шторм";

    Console.WriteLine(book3.Title); // теперь правильно
  }
}
```

При выполнении этой программы получается следующий результат.

```
Герберт Шилдт,  Полное руководство по C# 4.0, (c) 2010

Член book2.Title пуст.
Структура book2 теперь содержит:
Олдос Хаксли, О дивный новый мир, (c) 1932

Красный шторм
```

Как демонстрирует приведенный выше пример программы, структура может быть инициализирована с помощью оператора `new` для вызова конструктора или же путем простого объявления объекта. Так, если используется оператор `new`, то поля структуры инициализируются конструктором, вызываемым по умолчанию (в этом случае во всех полях устанавливается задаваемое по умолчанию значение), или же конструктором, определяемым пользователем. А если оператор `new` не используется,

как это имеет место для структуры book3, то объект структуры не инициализируется, а его поля должны быть установлены вручную перед тем, как пользоваться данным объектом.

Когда одна структура присваивается другой, создается копия ее объекта. В этом заключается одно из главных отличий структуры от класса. Как пояснялось ранее в этой книге, когда ссылка на один класс присваивается ссылке на другой класс, в итоге ссылка в левой части оператора присваивания указывает на тот же самый объект, что и ссылка в правой его части. А когда переменная одной структуры присваивается переменной другой структуры, создается копия объекта структуры из правой части оператора присваивания. Рассмотрим в качестве примера следующую программу.

Листинг 2

```
// Копировать структуру.

using System;

// Определить структуру.
struct MyStruct {
  public int x;
}

// Продемонстрировать присваивание структуры.
class StructAssignment
{
  static void Main()
  {
    MyStruct a;
    MyStruct b;

    a.x = 10;
    b.x = 20;

    Console.WriteLine("a.x {0}, b.x {1}", a.x, b.x);

    a = b;
    b.x = 30;

    Console.WriteLine("a.x {0}, b.x {1}", a.x, b.x);
  }
}
```

Вот к какому результату приводит выполнение этой программы.

```
a.x 10, b.x 20
a.x 20, b.x 30
```

Как показывает приведенный выше результат, после присваивания

```
a = b;
```

переменные структуры a и b по-прежнему остаются совершенно обособленными, т.е. переменная a не указывает на переменную b и никак не связана с ней, помимо того, что она содержит копию значения переменной b. Ситуация была бы совсем иной, если бы переменные a и b были ссылочного типа, указывая на объекты определенного класса. В качестве примера ниже приведен вариант предыдущей программы, где демонстрируется присваивание переменных ссылки на объекты определенного класса.

Листинг 3

```
// Использовать ссылки на объекты определенного класса.

using System;

// Создать класс.
class MyClass {
  public int x;
}

// Показать присваивание разных объектов данного класса.
class ClassAssignment
{
  static void Main()
  {
    MyClass a = new MyClass();
    MyClass b = new MyClass();

    a.x = 10;
    b.x = 20;

    Console.WriteLine("a.x {0}, b.x {1}", a.x, b.x);

    a = b;
    b.x = 30;

    Console.WriteLine("a.x {0}, b.x {1}", a.x, b.x);
  }
}
```

Выполнение этой программы приводит к следующему результату.

```
a.x 10, b.x 20
a.x 30, b.x 30
```

Как видите, после того как переменная `b` будет присвоена переменной a, обе переменные станут указывать на один и тот же объект, т.е. на тот объект, на который первоначально указывала переменная `b`.

1 О назначении структур

В связи с изложенным выше возникает резонный вопрос: зачем в C# включена структура, если она обладает более скромными возможностями, чем класс? Ответ на этот вопрос заключается в повышении эффективности и производительности программ. Структуры относятся к типам значений, и поэтому ими можно оперировать непосредственно, а не по ссылке. Следовательно, для работы со структурой вообще не требуется переменная ссылочного типа, а это означает в ряде случаев существенную экономию оперативной памяти. Более того, работа со структурой не приводит к ухудшению производительности, столь характерному для обращения к объекту класса. Ведь доступ к структуре осуществляется непосредственно, а к объектам – по ссылке, поскольку классы относятся к данным ссылочного типа. Косвенный характер доступа к объектам подразумевает дополнительные издержки вычислительных ресурсов на каждый такой доступ, тогда как обращение к структурам не влечет за собой подобные издержки. И вообще, если нужно просто сохранить группу связанных вместе данных, не требующих наследования и обращения по ссылке, то с точки зрения производительности для них лучше выбрать структуру.

Ниже приведен еще один пример, демонстрирующий применение структуры на

практике. В этом примере из области электронной коммерции имитируется запись транзакции. Каждая такая транзакция включает в себя заголовок пакета, содержащий номер и длину пакета. После заголовка следует номер счета и сумма транзакции. Заголовок пакета представляет собой самостоятельную единицу информации, и поэтому он организуется в отдельную структуру, которая затем используется для создания записи транзакции или же информационного пакета любого другого типа.

Листинг 4

```
// Структуры удобны для группирования небольших объемов данных.

using System;

// Определить структуру пакета.
struct PacketHeader
{
  public uint PackNum; // номер пакета
  public ushort PackLen; // длина пакета
}

// Использовать структуру PacketHeader для создания
// записи транзакции в сфере электронной коммерции.
class Transaction
{
  static uint transacNum = 0;

  PacketHeader ph;  // ввести структуру PacketHeader
                    // в класс Transaction
  string accountNum;
  double amount;

  public Transaction(string acc, double val) {
   // создать заголовок пакета
    ph.PackNum = transacNum++;
    ph.PackLen = 512;  // произвольная длина

    accountNum = acc;
    amount = val;
  }

  // Сымитировать транзакцию.
  public void sendTransaction()
  {
    Console.WriteLine("Пакет #: " + ph.PackNum +
                      ", Длина: " + ph.PackLen +
                      ",\n Счет #: " + accountNum +
                      ", Сумма: {0:C}\n", amount);
  }
}

// Продемонстрировать применение структуры в виде пакета
транзакции
class PacketDemo
{
  static void Main()
  {
    Transaction t = new Transaction("31243", -100.12);
    Transaction t2 = new Transaction("AB4655", 345.25);
```

```
        Transaction t3 = new Transaction("8475-09", 9800.00);

        t.sendTransaction();
        t2.sendTransaction();
        t3.sendTransaction();
    }
}
```

Вот к какому результату может привести выполнение этого кода.

```
Пакет #: 0, Длина: 512,
    Счет #: 31243, Сумма: $100.12

Пакет #: 1, Длина: 512,
    Счет #: AB4655, Сумма: $345.25

Пакет #: 2, Длина: 512,
    Счет #: 8475-09, Сумма: $9,800.12
```

Структура `PacketHeader` оказывается вполне пригодной для формирования заголовка пакета транзакции, поскольку в ней хранится очень небольшое количество данных, не используется наследование и даже не содержатся методы. Кроме того, работа со структурой `PacketHeader` не влечет за собой никаких дополнительных издержек, связанных со ссылками на объекты, что весьма характерно для класса. Следовательно, структуру `PacketHeader` можно использовать для записи любой транзакции, не снижая эффективность данного процесса.

Любопытно, что в C++ также имеются структуры и используется ключевое слово `struct`. Но эти структуры отличаются от тех, что имеются в C#. Так, в C++ структура относится к типу класса, а значит, структура и класс в этом языке практически равноценны и отличаются друг от друга лишь доступом по умолчанию к их членам, которые оказываются закрытыми для класса и открытыми для структуры. А в C# структура относится к типу значения, тогда как класс – к ссылочному типу.