Динамические структуры данных

Основные динамические структуры данных и их применение.

Õpikeskkond:	Eesti Ettevõtluskõrgkool Mainor e-õppe keskkond
Kursus:	BEST2012 IT-413 OO analüüs, disain ja programmeerimine (J. Faronova)
Raamat:	Динамические структуры данных
Printed by:	Juri Tretjakov
Date:	neljapäev, 13 detsember 2012, 17:35

Sisukord

1 Динамические структуры данных

В современных языках программировния, таких как C++, Java, C# и прочих, имеются средства создания динамических структур данных, которые позволяют во время выполнения программы образовывать объекты, выделять для них память, освобождать память, когда в них исчезает необходимость.

Статические типы данных требуют выделения памяти для хранения заранее, до начала работы с ними. Раздел оперативной памяти, распределяемый статически, называется статической памятью.

Для решения множества задач такой подход невозможен, как правило это задачи на обработку больших массивов оперативных данных. Если до начала работы с данными невозможно определить, сколько памяти потребуется для их хранения, память следует распределять во время выполнения программы по мере необходимости отдельными блоками. Блоки связываются друг с другом с помощью ссылок (указателей). Такой способ организации данных называется динамической структурой данных, поскольку ее размер изменяется во время выполнения программы, для этого структура размещается в динамической памяти. Динамической памятью (динамически распределяемой памятью) называется динамически распределяемый раздел памяти, .

Итак, **Динамические структуры данных** – это структуры данных, память под которые выделяется и освобождается по мере необходимости.

Динамические структуры данных в процессе существования в памяти могут изменять не только число составляющих их элементов, но и характер связей между элементами. При этом не учитывается изменение содержимого самих элементов данных. Такая особенность динамических структур, как непостоянство их размера и характера отношений между элементами, приводит к тому, что на этапе создания машинного кода программа-компилятор не может выделить для всей структуры в целом участок памяти фиксированного размера, а также не может сопоставить с отдельными компонентами структуры конкретные адреса. Для решения проблемы адресации динамических структур данных используется метод, называемый динамическим распределением памяти, то есть память под отдельные элементы выделяется в момент, когда они "начинают существовать" в процессе выполнения программы, а не во время компиляции. Компилятор в этом случае выделяет фиксированный объем памяти для хранения **указателя** - адреса динамически размещаемого элемента, а не самого элемента. Тип указатель является статическим, под него память выделяется во время компиляции, поэтому он требует описания. Значением указателя является адрес динамического объекта.

Итого, Динамическая структура данных характеризуется тем что:

- ей выделяется память в процессе выполнения программы;
- количество элементов структуры может не фиксироваться;
- отсутствует физическая смежность элементов структуры в памяти;
- размерность структуры может меняться в процессе выполнения программы;
- в процессе выполнения программы может меняться характер взаимосвязи между элементами структуры,

Необходимость в динамических структурах данных обычно возникает в следующих случаях:

- Используются переменные, имеющие довольно большой размер (например, массивы большой размерности), необходимые в одних частях программы и совершенно не нужные в других.
- В процессе работы программы нужен массив, список или иная структура, размер которой

изменяется в широких пределах и трудно предсказуем.
- Когда размер данных, обрабатываемых в программе, превышает объем сегмента данных.

Поскольку элементы динамической структуры располагаются по непредсказуемым адресам памяти, адрес элемента такой структуры не может быть вычислен из адреса начального или предыдущего элемента. Для установления связи между элементами динамической структуры используются **указатели**, через которые устанавливаются явные связи между элементами. Такое представление данных в памяти называется **связным**.

Преимущества связного представления данных – в возможности обеспечения значительной изменчивости структур:

- размер структуры ограничивается только доступным объемом машинной памяти;
- при изменении логической последовательности элементов структуры требуется не перемещение данных в памяти, а только коррекция указателей;
- большая гибкость структуры.

Вместе с тем, связное представление не лишено и недостатков, основными из которых являются следующие:

- на поля, содержащие указатели для связывания элементов друг с другом, расходуется дополнительная память;
- доступ к элементам связной структуры может быть менее эффективным по времени.

Последний недостаток является наиболее серьезным и именно им ограничивается применимость связного представления данных. Если в смежном представлении данных для вычисления адреса любого элемента нам во всех случаях достаточно было номера элемента и информации, содержащейся в дескрипторе структуры, то для связного представления адрес элемента не может быть вычислен из исходных данных. Дескриптор связной структуры содержит один или несколько указателей, позволяющих войти в структуру, далее поиск требуемого элемента выполняется следованием по цепочке указателей от элемента к элементу. Поэтому связное представление практически никогда не применяется в задачах, где логическая структура данных имеет вид вектора или массива – с доступом по номеру элемента, но часто применяется в задачах, где логическая структура требует другой исходной информации доступа (таблицы, списки, деревья и т.д.).

1.1 Классы динамических структур данных

К динамическим структурам относят:

- однонаправленные (односвязные) списки;
- двунаправленные (двусвязные) списки;
- циклические списки;
- стек;
- дек;
- очередь;
- бинарные деревья.

Они отличаются способом связи отдельных элементов и/или допустимыми операциями.

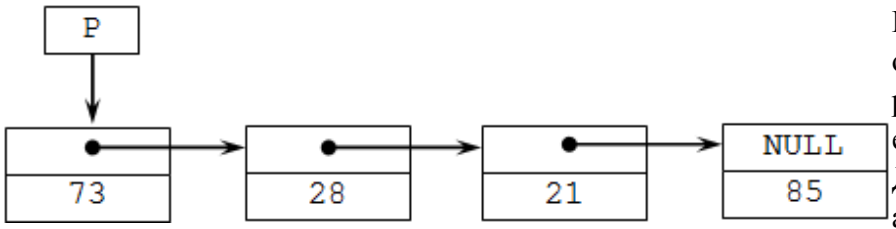
Поскольку элементами динамической структуры являются динамические переменные, то единственным средством доступа к динамическим структурам и их элементам является указатель (адрес) на место их текущего расположения в памяти. Таким образом, доступ к динамическим данным выполняется специальным образом с помощью указателей. Как правило это указатель на первый элемент структуры.

Строение динамической структуры

Каждый элемент динамической структуры состоит из двух полей:

- **информационного поля (поля данных)**, в котором содержатся те данные, ради которых и создается структура; в общем случае информационное поле само является интегрированной структурой – вектором, массивом, другой динамической структурой и т.п.;
- **адресного поля (поля связок)**, в котором содержатся один или несколько указателей, связывающий данный элемент с другими элементами структуры;

Количество информационных и адресных полей определяется классом структуры.



Рассмотрим например однонаправленный список, ниа рисунке слева приведена схема его организации в памяти. Данная структура использует адресное поле *Next,* для указания на следующий элемент.

Данная структура состоит из 4 элементов. Ее первый элемент имеет поле данных, равное 73, и связан с помощью своего поля *Next* со вторым элементом, поле данных которого равно 28, и так далее до последнего, четвертого элемента, поле данных которого равно 85, а поле *Next* равно *NULL*, то есть нулевому адресу, что является признаком завершения структуры.

Здесь **P** является указателем, который указывает на первый элемент структуры или на саму структуру.

Порядок работы с динамическими структурами данных следующий:

- создать (отвести место в динамической памяти);
- работать при помощи указателя;
- удалить (освободить занятое структурой место).

Создание экземпляра динамической структуры происходит при помощи того же ключевого слова ***new.***

1.2 Динамические структуры в C#

Динамические структуры данных рассмотренные выше в **.NET** называются **коллекциями**, соответсвующие классы находятся в пространстве `System.Collections` и `System.Collections.Generic`. Коллекции используются для хранения и обработки данных в течение цикла программы.

Примеры коллекций:

- Список, состоящий из фамилий сотрудников организации,
- набор целых чисел помещенных в массив,
- список дат в календаре,
- список строк в текстовом файле,
- множество объектов-представлений ячеек Excel документа.

Так как коллекции могут применяться для хранения совершенно различных элементов и для совершенно различных целей, важно правильно выбрать тот тип коллекции, который подойдет для решаемой задачи. От этого зависит скорость выполнения программы и объем используемых ресурсов. **.NET Framework** предлагает разработчику достаточно широкий набор реализованных коллекций, хотя и не содержит всех перечисленных выше типов, так, например, в нем нет системной реализации дек и деревьев.

Наиболее часто используемые коллекций .NET

Array (массив)

Позволяют хранить однотипные элементы. В памяти значения хранятся последовательно, размер памяти выделяемый под один элемент массива фиксирован и одинаков для всех элементов массива. Это свойство позволяет осуществлять быстрый доступ к элементу массива с помощью индекса. Размер массива задается перед его использованием, дальнейшее изменение размера связано с относительно высокими затратами.

List (список)

Список позволяет связанно хранить однотипные элементы, динамически выделяя память для последующих элементов. Это значит, что вам не нужно задавать размер списка при его инициализации в отличии от массивов. Доступ к элементу списка возможен по индексу, но обычно элементы списка обрабатываются последовательно.

Dictionary (словарь)

Словарь — это подходящая коллекция для хранения и последующего поиска однотипных элементов. В словаре элементы хранятся в виде пары «ключ-значение». Это значит, что вы можете сохранить значение, и в дальнейшем используя ключ быстро получить значение из словаря обратно. Тип ключа задается пользователем и может быть любым: числом, строкой или просто объектом.

ArrayList (список с разнотипными элементами)

Этот вид списка позволяет хранить разнотипные элементы. Это означает, что в одном и том же списке могут одновременно храниться как числа, структуры, так и строки, объекты и null значения. Что делает его значительно более медленным при поиске, чтении и записи элементов, чем его

более строго типизированный аналог — ***List***.

Hashtable (хэш-таблица)

По принципу работы хэш-таблица схожа с ***Dictionary***, за тем исключением, что позволяет одновременно использовать разнотипные ключи и хранить разнотипные элементы. Например для первого элемента хэш-таблицы можно использовать ключ в виде числового значения, а для второго, все той же таблицы, — в виде строки.

Tuple (кортеж)

Кортеж является «коллекцией для коллекции». Он позволяет хранить максимум до 6 разнотипных значений и призван ограничивать разработчика от рутинной работы по созданию классов-контейнеров.

BitArray (битовый массив)

Битовый массив применяется для работы с *Boolean* (логическими) значениями. Это значительно эффективнее, чем использовать любые другие коллекции, включая классические массивы.

Stack (стэк)

Стэк — это коллекция для организации хранения элементов по модели **LIFO** (*last-in-first-out*). Коллекция типа стэк незаменима для множества алгоритмов, среди наиболее известных — это алгоритм разбора (парсинга) **xml** документа.

Queue (очередь)

Очередь реализует другую модель добавления и удаления элементов — **FIFO** (*first-in-first-out*). Но в отличие от стэка очередь не так популярна, так как ту же функциональность может обеспечить коллекция *List* с лучшей производительностью. Очередь обычно используется для изящности кода при небольшой размере коллекции.

Функциональность коллекций *.NET*

Базовая функциональность коллекций в пространстве имен `System.Collections` предоставляется множеством интерфейсов.

- ***IEnumerable*** — предоставляет возможность проходить в цикле по элементам коллекции.
- ***ICollection*** — предоставляет возможность получать количество элементов в коллекции и копировать элементы в простой массив (унаследован от ***IEnumerable***).
- ***IList*** — предоставляет список элементов для коллекции с возможностями доступа к этим элементам и другими базовыми операциями работы со списками (унаследован от ***IEnumerable*** и ***ICollection***).
- ***IDictionary*** — подобен ***IList***, но предоставляет список элементов, доступных по значению ключа, а не по индексу (унаследован от ***IEnumerable*** и ***ICollection***).

1.3 Перебор объектов (интерфейс IEnumerable) и итераторы

Оператор `foreach` является удобным средством перебора элементов объекта. Массивы и все стандартные коллекции библиотеки ***.NET*** позволяют выполнять такой перебор благодаря тому, что в них реализованы интерфейсы *`IEnumerable`* и `IEnumerator`. Для применения оператора `foreach` к пользовательскому типу данных требуется реализовать в нем эти интерфейсы.

Интерфейс `IEnumerable` (*перечислимый*) определяет всего один метод — `GetEnumerator`, возвращающий объект типа `IEnumerator` (*перечислитель*), который можно использовать для просмотра элементов объекта.

Интерфейс `IEnumerator` задает три элемента:

- свойство `Current`, возвращающее текущий элемент объекта;
- метод `MoveNext`, продвигающий перечислитель на следующий элемент объекта;
- метод `Reset`, устанавливающий перечислитель в начало просмотра.

Цикл `foreach` использует эти методы для перебора элементов, из которых состоит объект.

Таким образом, в дальнейшем, если требуется, чтобы для перебора элементов класса мог применяться цикл `foreach`, необходимо реализовать четыре метода: `GetEnumerator`, `Current`, `MoveNext` и `Reset`.

Это неинтересная работа, а выполнять ее приходится часто, поэтому уже в версию 2.0 были введены средства, облегчающие выполнение перебора в объекте — **итераторы**.

Итератор представляет собой метод, задающий последовательность перебора элементов объекта. На каждом проходе цикла `foreach` выполняется один шаг итератора, заканчивающийся выдачей очередного значения. Выдача значения выполняется с помощью ключевого слова `yield`.

Попросту говоря, итератором называется такой член, который указывает, каким образом должны возвращаться внутренние элементы контейнера при обработке в цикле `foreach`. Хотя метод итератора должен все равно носить имя GetEnumerator (), а его возвращаемое значение относиться к типу IEnumerator, необходимость в реализации каких-либо ожидаемых интерфейсов в специальном классе при таком подходе отпадает.

Пример:

```
public class Garage {
// для некоторго класса Car
private Car[] carArray = new Car[4];

// Метод итератора.
public IEnumerator GetEnumerator () {
 foreach (Car c in carArray) {
 yield return c;
 }
}
}
```

В данной реализации GetEnumerator () проход по подэлементам осуществляется с использованием внутренней логики foreach, а каждый объект Car возвращается вызывающему коду с применением синтаксиса yield return. При достижении оператора yield return производится сохранение текущего

местоположении в контейнере, и при следующем вызове итератора выполнение начинается уже с этого места.

Преимущество использования итераторов заключается в том, что для одного и того же класса можно задать различный порядок перебора элементов. Для этого используются, так называемые **именованные итераторы**.

Создание именованного итератора

Ключевое слово yield формально может применяться внутри любого метода, как бы ни выглядело его имя. Такие методы (называемые **именованными итераторами**) уникальны тем, что могут принимать любое количество аргументов, **НО** необходимо очень хорошо понимать, что **метод будет возвращать интерфейс IEnumerable**, а не ожидаемый совместимый с IEnumerator тип.

Для примера добавим к типу Garage следующий метод:

```
public IEnumerable GetReversedCars() {
// Возврат элементов в обратном порядке.
for (int i = carArray. Length; i != 0; i--)
 yield return carArray[i-l];
}
```

Теперь с ним можно взаимодействовать следующим образом:

```
static void Main(string[ ] args) {
// описание класса выше
Garage carLot = new Garage ();

// Получение элементов с помощью обычного способа GetEnumerator().
foreach (Car c in carLot){
 //Обработать Car c, например
  Console.WriteLine("{0} is going {1} MPH", c.Name, c.CurrentSpeed);
}

Console.WriteLine();

// Получение элементов (в обратном порядке) с помощью именованного итератора.
foreach (Car c in carLot.GetReversedCars() ) {
 //Обработать Car c
  Console.WriteLine("{0} is going {1} MPH", c.Name, c.CurrentSpeed);
}

Console.ReadLine() ;
}
```

Именованные итераторы представляют собой очень полезные конструкции, поскольку позволяют определять в единственном специальном контейнере сразу несколько способов для запрашивания возвращаемого набора.

1.4 Коллекция ArrayList и ее применение

Коллекция **ArrayList** моделирует поведение массива, с динамическим размером. Рассмотрим основные приёмы работы с этой коллекцией.

Создание объекта класса ArrayList

Создания объекта класса **ArrayList** выполняется практически так же, как создаётся объект любого другого класса. Рассмотрим основные варианты создания объектов класса **ArrayList**:

1) создаём объект x типа **ArrayList**, который не содержит ни одного элемента:

```
ArrayList x = new ArrayList();
```

Это наиболее распространённый метод создания объекта типа **ArrayList**. В этом случае объект имеет по-умолчанию ёмкость 16 элементов, т.е. память не будет выделяться дополнительно до тех пор, пока в объекте находится не более 16 элементов.

2) создаём объект x типа **ArrayList**, который рассчитан на хранение заданного количества элементов (n):

```
int n = 20;
ArrayList x = new ArrayList(n);
```

Здесь под массив будет выделено памяти не менее, чем под n элементов.

Добавление новых элементов в массив в конец массива

Для добавление новых элементов в массив типа **ArrayList** имеется два метода. В обоих случаях **добавление всегда делается в конец данного массива**:

1) Для добавления новых элементов в массив типа **ArrayList** по одному используется метод **Add**():

```
x.Add(100);
```

Здесь в конец массива x добавляется число 100. К стати, метод **Add**() возвращает значение целого типа, являющееся номером добавляемого элемента. На практике этот номер обычно не используется.

2) Добавление в конец массива ранее созданного массива:

```
x1.AddRange(x);
```

Теперь в массиве x1 содержатся не только его прежние элементы, но и новые, скопированные из массива x. Сам же массив x остаётся без изменения.

Удаление данных из массива

Для удаления данных также имеется несколько вариантов:

1) Удаление всех элементов массива (очистка массива):

```
x.Clear();
```

После выполнения такого оператора массив x типа **ArrayList** станет пустым. Сам объект x не удаляется!

2) Удаление заданного элемента массива (надо указать значение того, что ищем, и, если элемент с таким значением будет найдет, то он удалится):

```
x.Remove(100);
```

3) Удаление одного элемента, заданного своим номером:

```
int i = 1;
x1.RemoveAt(i);
```

4) Удаление нескольких подряд идущих элементов:

```
int i = 2, k = 3;
x.RemoveRange(i, k);
```

Здесь из массива x, начиная с позиции i (2), будут удалены k (3) элементов.

Определение размера массива

Имеется два удобных свойства:

1) **Count** — определяет текущее количество элементов в массиве. Пример типичного использования этого свойства (в цикле выводим элементы массива на печать):

```
for(int i = 0; i < x.Count; i++)   Console.WriteLine(x[i]);
```

2) **Capacity** — хранит ёмкость объекта типа **ArrayList**, т.е. максимальное количество элементов, которое может храниться в объекте без дополнительного выделения памяти.

Сортировка массива

Во многих случаях бывает удобным работать с упорядоченным массивом данных. Для этого используется метод **Sort**():

1) упорядочение всего массива:

```
x.Sort();
```

2) упорядочение данного количества элементов массива, начиная с заданного элемента:

```
x.Sort(1, 2, null);
```

Здесь мы упорядочиваем 2 элемента массива, начиная с 1-го (индексы начинаются с 0). Третий параметр, имеющий значение ***null***, означает, что у нас используется стандартный *компаратор* (объект, производящий сравнение). Если сортируется массив, состоящий из объектов стандартных типов (как в нашем случае), то компаратор не требуется перегружать. Если сортировать массив, состоящий из объектов класса (например, ***Car***), созданного пользователем, то требуется реализовать в этом классе (***Car***) интерфейсы ***Icomparable*** и ***IComparer***, а ссылку на реализованный интерфейс ***IComparer*** должна быть передана методу **Sort**() в качестве третьего параметра.

3) упорядочение всего массива, состоящего из объектов класса пользователя:

```
x.Sort(t);
```

где t — ссылка на реализованный интерфейс ***IComparer*** (см. выше). Реализация интерфейсов ***IComparer*** и ***Icomparable*** было рассмотрено в теме 2. Наследование.

Расположение элементов массива в обратном порядке

Используется метод **Reverse**(), имеющий две реализации:

1) Применить метод для всего массива:

```
x.Reverse();
```

2) Записать требуемое количество элементов (у нас их 3) в обратном порядке, начиная заданной позиции (у нас это 2):

```
x.Reverse(2,3);
```

NB! Метод **Sort**() упорядочивает массив по возрастанию. Если после вызова метода **Sort**() применить ещё и **Reverse**(), то получим массив, упорядоченный по убыванию.

1.5 Коллекция Queue и ее применение

Структура *Queue*

Очередь (*queue*) - это класс коллекции, организованный по принципу **FIFO** (первым вошел - первым вышел). Классическая аналогия - очередь в кассу за билетами. Первый человек, стоящий в очереди, первым и выйдет из нее, когда купит билет.

Очередь удачно подходит для управления ограниченными ресурсами.

Создание объекта класса *Queue*

Создания объекта класса **Queue** выполняется практически так же, как создаётся объект любого другого класса. Рассмотрим основные варианты создания объектов класса **Queue**:

1) создаём объект x типа ***Queue***, который не содержит ни одного элемента:

```
Queue x = new Queue();
```

Это наиболее распространённый метод создания объекта типа **Queue**.

2) создаём объект x типа **Queue** , который рассчитан на хранение заданного количества элементов (*n*):

```
int n = 20;
Queue x = new Queue(n);
```

Здесь под очередь будет выделено памяти не менее, чем под n элементов.

3) создаём объект x типа **Queue** , который заполняется элементами коллекции **Col**.

```
Queue x = new Queue(Col);
```

В этом случае создается новый экземпляр класса **Queue**, содержащий элементы, скопированные из указанной коллекции, и обладающий начальной емкостью, равной количеству скопированных элементов.

Добавление новых элементов в очередь

Для добавление новых элементов в **очередь** имеется метод Enqueue(). **Добавление всегда делается в конец очереди**:

Для добавления новых элементов в **Queue** по одному используется метод **Enqueue**():

```
x.Enqueue(100);
```

Здесь в конец очереди x добавляется число 100.

Удаление данных из очереди

Для удаления данных также имеется несколько вариантов:

1) Удаление всех элементов очереди (очистка очереди):

```
x.Clear();
```

После выполнения такого оператора очередь станет пустой. Сам объект x не удаляется!

2) Удаление следующего элемента очереди для последующей обработки :

```
Object o = x.Dequeue();
```

3) Иногда необходимо получить объект без удаления его из очереди (например для контроля). В таком случае используется метод **Peek**().

```
Object o = x.Peek();
```

Определение размера очереди

Имеется специальное свойство **Count** — определяет текущее количество элементов в очереди.

```
int c = x.Count;
```

Емкость коллекции **Queue** всегда больше или равна значению свойства *Count*. Если значение свойства *Count* при добавлении элементов превысит значение емкости, то перед копированием старых элементов и добавлением новых емкость автоматически увеличивается посредством перераспределения внутреннего массива. Новая емкость определяется с помощью умножения текущей емкости на коэффициент роста, который определяется при построении коллекции **Queue**. Емкость коллекции **Queue** всегда увеличивается не меньше, чем на 4, независимо от фактора роста; значение фактора роста, равное 1.0, не предотвращает увеличения размера коллекции **Queue** .

Емкость может быть уменьшена посредством вызова метода *TrimToSize*().

Проверка наличия элемента в очереди

Для проверки наличия объекта в очереди используется метод **Contains**(). Данный метод определяет равенство с помощью вызова метода Object.Equals().

С помощью этого метода выполняется линейный поиск; поэтому данный метод является операцией O(n), где n— значение свойства *Count*.

Начиная с версии **.NET Framework 2.0** данный метод использует методы *Equals()* и *CompareTo()* класса объектов из очереди чтобы определить существование элемента. В предыдущих версиях **.NET Framework** такое определение осуществлялось путем применения к объектам коллекции методов *Equals()* и *CompareTo()* с параметром *obj*.

```
// Object item - искомый элемент
if (x.Contains(item))
//...Некие действия
```

Метод возвращает соответственно значение *true* или *false*.

Преобразование в массив

Для преобразования очереди в массив существует 2 метода: ***CopyTo***() и ***ToArray***().

1) В случае использования метода ***CopyTo***() элементы коллекции **Queue** копируюстя в существующий одномерный массив **Array**, начиная с указанного значения индекса массива. Существующие по указанным индексам в массиве элементы при этом переопределяются.

```
Queue mySourceQ = new Queue();
Array myTargetArray=new Array(15);
mySourceQ.CopyTo( myTargetArray, 6 );
```

2) В случае использования метода ***ToArray***() элементы коллекции **Queue** копируюстя в новый массив.

```
Queue mySourceQ = new Queue();
Object[] myStandardArray = mySourceQ.ToArray();
```

1.6 Коллекция Stack и ее применение

Структура Стек

Стек (*Stack*) - это класс коллекции, организованный по принципу **LIFO** (последним вошел - первым вышел). Классическая аналогия - стопка тарелок. Верхняя тарелка всегда будет использована первой, в то время как для изъятия тарелок из середины, необходимо сначала снять верхние.

Добавление элемента, называемое также проталкиванием (**push**), возможно только в вершину стека (добавленный элемент становится первым сверху). Удаление элемента, называемое также выталкиванием (**pop**), тоже возможно только из вершины стека, при этом второй сверху элемент становится верхним.

Стеки широко применяются в вычислительной технике. Например, для отслеживания точек возврата из подпрограмм используется **стек вызовов**, который является неотъемлемой частью архитектуры большинства современных процессоров. Языки программирования высокого уровня также используют стек вызовов для передачи параметров при вызове процедур.

Создание объекта класса *Stack*

Создания объекта класса **Stack** выполняется практически так же, как создаётся объект любого другого класса. Рассмотрим основные варианты создания объектов класса **Stack**:

1) создаём объект x типа ***Stack***, который не содержит ни одного элемента:

```
Stack x = new Stack();
```

Это наиболее распространённый метод создания объекта типа ***Stack***.

2) создаём объект x типа ***Stack***, который рассчитан на хранение заданного количества элементов (*n*):

```
int n = 20;
Stack x = new Stack(n);
```

Здесь под стек будет выделено памяти не менее, чем под ***n*** элементов.

3) создаём объект x типа ***Stack***, который заполняется элементами коллекции **Col**.

```
Stack x = new Stack (Col);
```

В этом случае создается новый экземпляр класса ***Stack***, содержащий элементы, скопированные из указанной коллекции, и обладающий начальной емкостью, равной количеству скопированных элементов.

Добавление новых элементов в стек

Для добавление новых элементов в **стек** имеется метод ***Push***(). **Добавление всегда делается в начало стека.**

```
x.Push(100);
```

Здесь в начало (верхушку) стека x добавляется число 100.

Удаление данных из стека

Для удаления данных также имеется несколько вариантов:

1) Удаление всех элементов стека (очистка стека):

```
x.Clear();
```

После выполнения такого оператора стек будет пустой. Сам объект x не удаляется!

2) Удаление следующего элемента стека для последующей обработки :

```
Object o = x.Pop();
```

3) Иногда необходимо получить объект без удаления его из стека (например для контроля). В таком случае используется метод **Peek**().

```
Object o = x.Peek();
```

Определение размера стека

Имеется специальное свойство **Count** — определяет текущее количество элементов в стеке.

```
int c = x.Count;
```

Емкость всегда больше или равна значению свойства ***Count***. Если значение ***Count*** превысит емкость при добавлении элементов, емкость автоматически увеличивается посредством перераспределения внутреннего массива перед копированием старых элементов и добавлением новых.

Проверка наличия элемента в стеке

Для проверки наличия объекта в очереди используется метод ***Contains***(). Данный метод определяет равенство с помощью вызова метода *Object.Equals()*.

С помощью этого метода выполняется линейный поиск; поэтому данный метод является операцией O(n), где n— значение свойства *Count*.

Начиная с **.NET Framework 2.0** данный метод определяет наличие элемента в коллекции с помощью методов *Equals()* и *CompareTo()* с параметром *item*. В предыдущих версиях **.NET Framework** такое определение осуществлялось путем применения к объектам коллекции методов *Equals()* и *CompareTo()* с параметром *item*.

```
// Object item - искомый элемент
if (x.Contains(item))
//...Некие действия
```

Метод возвращает соответственно значение *true* или *false*.

Преобразование в массив

Для преобразования стека в массив существует 2 метода: ***CopyTo***() и ***ToArray***().

1) В случае использования метода ***CopyTo***() элементы коллекции **стека** копируюстя в существующий одномерный массив **Array**, начиная с указанного значения индекса массива. Существующие по указанным индексам в массиве элементы при этом переопределяются.

```
Stack myStack = new Stack();
Array myTargetArray=new Array(15);
myStack.CopyTo( myTargetArray, 6 );
```

2) В случае использования метода ***ToArray***() элементы коллекции **стека** копируюстя в новый массив.

```
Stack myStack = new Stack();
Object[] myStandardArray = myStack.ToArray();
```

1.7 Коллекция HashTable и ее применение

Структура HashTable

Хеш-таблица — это структура данных, реализующая интерфейс ассоциативного массива, а именно, она позволяет хранить **пары** (**ключ, значение**) и выполнять три операции: операцию добавления новой пары, операцию поиска и операцию удаления пары по ключу.

Класс *HashTable* предоставляет коллекцию пар "ключ-значение", которые упорядочены по хэш-коду ключа.

Свойства хеш-таблицы

Каждый элемент в паре "ключ-значение" хранится в объекте DictionaryEntry. Ключ не может быть равным *null*, а значение — может.

Важное свойство хеш-таблиц состоит в том, что, при некоторых разумных допущениях, все три операции (поиск, вставка, удаление элементов) в среднем выполняются за время *O(1)*. При добавлении элемента в коллекцию *Hashtable* он помещается в определенный сегмент в зависимости от хэш-кода ключа. В дальнейшем поиск ключа осуществляется только в определенном сегменте с использованием хэш-кода ключа. Таким образом в значительной степени уменьшается количество операций сравнения ключей, которое требуется для нахождения элемента.

Но при этом не гарантируется, что время выполнения отдельной операции мало. Это связано с тем, что при достижении некоторого значения коэффициента заполнения необходимо осуществлять перестройку индекса хеш-таблицы: увеличить значение размера массива и заново добавить в пустую хеш-таблицу все пары.

Показатель загрузки коллекции *Hashtable* определяет максимальное отношение количества элементов к количеству сегментов. Снижение показателя загрузки уменьшает среднее время поиска за счет увеличения объема используемой памяти. Значение показателя загрузки по умолчанию, равное 1.0, обычно обеспечивает наилучшее соотношение между объемом памяти и временем поиска. При создании коллекции *Hashtable* может быть задан другой показатель загрузки.(См. ниже)

Когда элементы добавляются в коллекцию *Hashtable*, фактический показатель загрузки коллекции *Hashtable* увеличивается. Когда фактический показатель загрузки становится равным заданному, количество сегментов в объекте *Hashtable* автоматически увеличивается до наименьшего простого числа, которое больше удвоенного текущего количества сегментов в коллекции *Hashtable*.

Каждый ключевой объект в коллекции *Hashtable* должен иметь свою собственную хэш-функцию, доступ к которой может быть получен при вызове метода *GetHash*.

Методы класса *HashTable*

Основные методы класса *HashTable* перечислены в Таблице 1.

Таблица 1. Основные методы класса *HashTable*

Метод	Описание
	Инициализирует новый пустой экземпляр класса *Hashtable* с заданными по

Hashtable()	умолчанию начальной емкостью, показателем загрузки, поставщиком хэш-кода и объектом сравнения.
Hashtable(IDictionary)	Инициализирует новый пустой экземпляр класса Hashtablepосредством копирования элементов из указанного словаря в новый объект Hashtable. У нового объекта Hashtable исходная емкость равна числу копируемых элементов, и он обладает заданными по умолчанию показателем загрузки, поставщиком хэш-кода и объектом сравнения.
Hashtable(Int32)	Инициализирует новый пустой экземпляр класса Hashtable с указанной исходной **емкостью** и заданными по умолчанию показателем загрузки, поставщиком хэш-кода и объектом сравнения.
Add	Добавляет элемент с указанными ключом и значением в коллекциюHashtable.
Clear	Удаляет все элементы из коллекции Hashtable.
Contains, ContainsKey	Определяет, содержит ли коллекция Hashtable указанный ключ.
ContainsValue	Определяет, содержит ли коллекция Hashtable указанное значение.
GetHash	Возвращает хэш-код указанного ключа.
KeyEquals	Сравнивает указанный объект класса Object с указанным ключом, который содержится в коллекции Hashtable.
Remove	Удаляет элемент с указанным ключом из коллекции Hashtable.

Использование класса *HashTable*

В операторе **foreach** языка C# необходима информация о типе каждого элемента коллекции. Так как каждый элемент коллекции *Hashtable* представляет собой пару "ключ-значение", тип элемента не является типом ключа или типом значения. Вместо этого в качестве типа элемента используется DictionaryEntry.

Например:

```
Hashtable myHashtable = new Hashtable();

//первый элемент - ключ, второй - значение
myHashtable.Add(1, "Book 1");
myHashtable.Add(2, "Book 2");
myHashtable.Add(3, "Book 3");
myHashtable.Add(4, "Book 4");

//При повторной попытке добавить существующий ключ будет сгенерировано исключение

foreach (DictionaryEntry de in myHashtable) {
 Console.WriteLine("Key = {0}, Value = {1}", de.Key, de.Value);
}

if (MyHashTable.ContainsKey(2)) myHashTable.Remove(2);
```

Свойства класса *HashTable*

Класс имеет множетсво свойств, наиболее часто используемые из которых:

- *Count* - Получает число пар "ключ-значение" в коллекции *Hashtable*.
- *Keys* - Получает коллекцию *ICollection*, содержащую **ключи** из коллекции *Hashtable*.
- *Values* - Получает коллекцию *ICollection*, содержащую **значения** из коллекции *Hashtable*.

- Свойство *Item* можно также использовать для добавления новых элементов посредством задания значения ключа, которого не существует в коллекции *Hashtable*.

Использование *Item*.

Например

```
myHashTable[5] = "Book 5".
```

Однако, если указанный ключ уже существует в коллекции *Hashtable*, задание свойства *Item* перезаписывает прежнее значение. Напротив, метод Add не изменяет существующих элементов.

2 Заключение

В данной главе были рассмотрены основные динамические структуры данных и их реализации на языке C#.

- **Динамические структуры** данных это структуры, память под которые выделяется и освобождается по мере необходимости.
- Динамические структуры данных в процессе существования в памяти могут изменять не только число составляющих их элементов, но и характер связей между элементами.
- Динамические структуры состоят из ссылок - указателей на объекты классов членов.
- **Указатель** - адрес в памяти динамически размещаемого элемента.
- Динамические структуры используюстя когда
    - Используются переменные, имеющие довольно большой размер (например, массивы большой размерности), необходимые в одних частях программы и совершенно не нужные в других.
    - В процессе работы программы нужен массив, список или иная структура, размер которой изменяется в широких пределах и трудно предсказуем.
    - Когда размер данных, обрабатываемых в программе, превышает объем сегмента данных.
- Основными динамическими структурами являются Списки, Очереди, Стеки, Хеш-таблицы, Деревья.
- Динамические структуры данных в **.NET** называются **коллекциями**, соответсвующие классы находятся в пространстве *System.Collections* и *System.Collections.Generic*.

- Базовая функциональность коллекций в пространстве имен *System.Collections* предоставляется множеством интерфейсов: ***IEnumerable, ICollection, IList, IDictionary***.

3 Дополнительное чтение и использованные материалы.

Дополнительный материал для изучения:

- *Джесс Либерти,* **Программирование на C# (2-е издание)**, 2003. Изучить 1 Часть. Глава 9.

- *Нейгел К., Ивьен Б., Глинн Дж., Уотсон К.* **C# 4.0 и платформа .NET4 для профессионалов.** Изучить *Главы 3, 6* и *10, 29* (стр 893)

- *Балена Ф. и Димауро Д.* - **Современная практика программирования на Microsoft Visual Basic и Visual C#,** - 2006. Изучить Главу 22.

Использованные материалы

Документация MSDN. Пространство имен System.Collections

Балена Ф. и Димауро Д. - **Современная практика программирования на Microsoft Visual Basic и Visual C#,** - 2006

Джесс Либерти, **Программирование на C# (2-е издание)**, 2003.

Нейгел К., Ивьен Б., Глинн Дж., Уотсон К. **C# 4.0 и платформа .NET4 для профессионалов.**