УДК 681.3

А.М. Ноткин

Пермский государственный технический университет

ЭВОЛЮЦИЯ КОЛЛЕКЦИЙ В ЯЗЫКАХ ПРОГРАММИРОВАНИЯ

Рассматривается структура хранения наборов данных, начиная от массивов и заканчивая классами коллекций в наиболее распространенных языках программирования – Pascal, C++, Java, C#. На основе их анализа приводятся рекомендации по использованию коллекций для хранения и обработки данных.

Под коллекцией будем понимать элементы языка, позволяющие хранить и обрабатывать данные одного или родственных типов.

Простейшим видом коллекций является массив. Массивы в той или иной форме поддерживают все императивные языки программирования. Однако возможности, предоставляемые массивами, менялись по мере развития ЯП.

1. Поддержка массивом в языке Pascal. В классическом Pascal Н. Вирта имеется тип array, который определяет тип массива с заданными границами индексов, например:

```
Const n=20;
Type vector=array[1..n] of integer;
Var a:vector;
```

Память для массива выделяется автоматически в соответствии с описанием (типом массива). В приведенном выше примере для массива «а» выделяется память размером 20*sizeof(integer).

Недостаток таких массивов очевиден – нельзя написать функцию (процедуру), которой можно передать массив любой длины. Также массивы с одним типом элементов, но имеющие разный размер, несовместимы.

В реализации Turbo Pascal фирмы «Borland» введены динамические массивы, для которых необходимая память выделяется во время выполнения программы. Тип таких массивов определяется без задания конкретных границ индексов, например:

```
Type vector=array of integer;
Var a:array;
```

При определении переменной массива «a» память не выделяется. Память выделяется динамически процедурой SetLength():

```
SetLength(a,n);
```

В этом случае можно написать процедуру:

```
Procedure MyProc(a:array of integer)
begin
//работа с массивом a
end;
```

Определив два массива:

```
Var a,b:array of integer;
SetLength(a,10); SetLength(b,20);
```

Для каждого из них можно вызвать одну и ту же процедуру MyProc():

```
MyProc(a); MyProc(b);
```

Для получения границ индексов используются функции Low(a) и High(a). Выход индексов за границу можно контролировать, если включить опцию компилятора Range checking.

При присваивании a:=b копируются ссылки (указатели). Для копирования значений используется функция Copy(): a:=Copy(b).

Если определить массив указателей на объекты, то получим полиморфный массив, который можно использовать для хранения родственных объектов. При наличии в классах этих объектов виртуальных функций можно, получив доступ к указателю на *i*-й объект, вызвать виртуальную функцию класса *i*-го объекта. Такая возможность является следствием того, что ссылка (указатель) типа базового класса может указывать как на объект этого, так и любого производного класса.

2. Поддержка массивов в языках C/C++. В C/C++ нет встроенного типа массива, как array в Pascal. Когда мы определяем массив int a[10], либо int* a=new int[10], в обоих случаях имя «a» обозначает просто константный указатель на начало непрерывной области памяти, к которой можно обращаться по индексу a[i]. Определение int[10] просто резервирует область для 10 целых чисел. Обращение a[i] со значением I меньше 0 и больше 9 не является синтаксической ошибкой, но может привести к ошибкам времени выполнения.

Адрес *i*-го элемента в C определяется с использование адресной арифметики, как *(a+i). Такой прямой доступ к памяти в C эффекти-

вен, по порождает ряд проблем. Об одной сказано выше – нет контроля выхода индекса за границу. Вторая требует при передаче массива в функцию дополнительного параметра для указания длины массива.

Например:

```
void  f(int* a, int n);
```

где *a* – указатель на начало массива, *n* – размер массива.

Хотя передача в функцию массива требует дополнительного параметра, но зато в функцию можно передать массив любой длины.

Так же, как и в языке Pascal, в C, если определить массив указателей на объекты, то получим полиморфный массив, который можно использовать для хранения родственных объектов.

3. Поддержка массивов в языке Java. В отличие от C в Java объявление тип[] вводит массив элементов. Этот массив рассматривается как класс, наследуемый от Object, что существенно отличает их от массивов в C. В классе массива Java определено поле public final int length, хранящее длину (количество элементов) массива.

Например, int[] a={1,2,3,4,5} – это массив из 5 целых чисел, aint[]b={1,2,3,4,5,6,7,8} – массив из 8 целых чисел. Длину (количество элементов) массива можно определить с помощью length. Например, a.length возвратит 5, a b.length – 8.

Таким образом, в Java с массивами можно работать как с объектами определенного типа. Можно передавать их в функции как параметры. Выход индекса за границы генерирует исключение ArrayIndexOutOfBoundsException.

Если *a* и *b* – массивы одного типа, то *a* = *b* означает, что *a* и *b* ссылаются на один и тот же массив. Если требуется скопировать значения, то следует использовать функцию clone() – a=b.clone().

4. Поддержка массивов в языке C#. Хотя массивы в языке C# можно использовать так же, как и в других ЯП, они имеют важную особенность. Дело в том, что в C# массивы являются объектами. Базовым классом для всех массивов является класс Array. В этом классе определено множество методов, которые позволяют выполнять большинство типовых операций над массивами: сортировка, поиск, копирование, преобразование.

Например, для сортировки имеется ряд перегруженных методов Sort(), позволяющих сортировать весь массив, его часть или два массива, которые содержат соответствующие пары ключ/значение. При сортировке может быть задан критерий (функция сравнения).

Поиск может выполняться как в неотсортированном массиве (группа методов Find()), так и в отсортированном (методы BinarySearch()), причем поиск может выполняться как по значению, так и по условию.

Представляет интерес метод ForEach(), который выполняет некоторое действие над каждым элементом массива.

Начиная с версии C# 2.0, добавлены обобщенные методы (шаблоны функций). Использование обобщенных методов обеспечивает типовую безопасность, так как тип аргументов задается явно. Таким образом, не требуется выполнения приведения типа.

Все методы класса Array доступны для всех встроенных типов значений. Для массивов объектов при выполнения операций, требующих сравнение элементов (например, сортировка и поиск), класс этих объектов должен реализовать интерфейс IComparable или IComparable<T>. Это, впрочем, не составляет особого труда, так как надо реализовать всего лишь один метод сравнения int CompareTo(object obj).

В C# имеется оператор цикла foreach, который позволяет осуществить последовательный доступ к элементам коллекции, в том числе и массива. Проходя по массиву, мы получаем значения каждого его элемента в так называемой итераторной переменной. Однако итераторная переменная доступна только для чтения, изменить с ее помощью элементы массива нельзя. Однако если элементом массива является объект, можно изменить его поля. Аналогично и для метода Array.ForEach().

Как видно, даже на уровне обычных массивов C# предоставляет достаточно мощную поддержку в процессах хранения и обработки данных.

5. Контейнеры стандартной библиотеки шаблонов (STL) C++. Гораздо большие возможности для хранения данных предоставляют специальные классы – коллекции.

Коллекции представляют собой реализации абстрактных структур данных, поддерживающих три основные операции:

– добавление элемента в коллекцию;

– удаление элемента из коллекции;

– изменение элемента в коллекции.

Коллекции в языках программирования можно разделить на коллекции, построенные на основе шаблонов (обобщений) и нешаблонные. Первые позволяют при создании задавать тип хранимых объектов, вторые хранят объекты определенного для них типа.

Другая классификация делит коллекции на последовательные и ассоциативные.

Последовательные коллекции:

– динамические массивы;
– списки;
– очереди;
– стеки.

Ассоциативные коллекции хранят пары ключ/значение и позволяют по ключу получить доступ к значению.

Ассоциативные коллекции:

– карты;
– хэш-таблицы.

В той или иной степени все языки программирования в своих библиотеках поддерживают эти типы коллекций.

В библиотеке STL C++ коллекции называются контейнерами и наряду с итераторами и алгоритмами составляют ядро библиотеки. Все контейнеры построены на основе шаблонов (template). Это следующие контейнеры:

– последовательные: динамический массив vector, список list, список с двумя концами deque;

– ассоциативные: карта map и мультикарта multymap, множество set и мультимножество multyset;

– адаптеры контейнеров: очередь queue, стек stack, очередь с приоритетом priority_queue;

Для каждого класса контейнера определен класс итераторов, позволяющий получать простой доступ к элементам контейнера. Итераторы работают с контейнером так же, как указатели с массивами.

Контейнеры имеют несколько перегруженных конструкторов, в том числе конструктор, позволяющий инициализировать контейнер с массивом.

Например, для контейнера vector:

```
int a[]={5,8,2,1,-6,4,8,-2,8};
vector<int> v(a,a+9);
```

Библиотека STL организована таким образом, что функциональность контейнеров сведена к минимуму, их функция – только хранить объекты и обеспечить эффективный доступ к ним. Для работы же с контейнерами предназначены специальные шаблоны функций – алгоритмы.

Контейнер передается в алгоритм как последовательность с двумя параметрами – итераторами на начало и конец последовательности. Поэтому алгоритмы успешно работают и с обычными массивами С.

Например, можно очень просто отсортировать массив:

```
int a[9]={5,8,2,-1,-6,4,8,-2,8};
sort(a,a+9);
```

Контейнер вектор можно отсортировать так:

```
sort(v.begin(),v.end());
```

Для работы с контейнерами в STL реализованы типовые и широко используемые алгоритмы, такие как сортировка, поиск, перемещение, замена, выполнение заданных действий над всеми элементами.

6. Коллекции в Delphi. Коллекции Delphi являются частью библиотеки визуальных компонентов VCL и служат в основном для хранения данных в визуальных компонентах Delphi, многие из которых содержат свойства типа коллекций. Например, компоненты TListBox и TComboBox имеют свойство Items:TStrings, где TStrings – коллекция строк и связанных с ними объектов.

Однако эти коллекции могут использоваться и самостоятельно.

Коллекция TStringList. Класс TStrings имеет потомка TStringList, который представляет собой динамический массив записей типа TStringItem.

```
TStringItem=record
FString:string;
FObject:TObject;
end;
```

Эта запись объединяет строку string и связанный с ней объект TObject. Таким образом, TStringList можно рассматривать как ассоциативный контейнер, содержащий пару ключ/значения. Но если тип значения может любым, наследуемым от TObject, то тип ключа только string. Это является ограничением по сравнению с ассоциативными контейнерами в других библиотеках, например, STL C++, библиотеках Java и C#.

Векторные свойства Strings и Objects позволяют получить доступ к строке и объекту по индексу в массиве.

Прямого метода, возвращающего объект по ассоциированной с ним строке, у TStringList нет, для этого можно использовать метод IndexOf() и свойство Objects.

Следующий простой пример показывает использование коллекции TStringList.

```
Type
Person=class(TObject)
  protected
  name:string;
  age:integer;
  public
  Constructor Create(name1:string;age1:integer);
  procedure Show();virtual;
end;
Student=class(Person)
  protected
  grade:real;
  public
  Constructor                                    Cre-
ate(name1:string;age1:integer;grade1:real);
  procedure Show();override;
  end;
Var
p1,p2:Person;
list:TStringList;
begin
p1:=Person.Create('Ivanov',25);
p2:=Student.Create('Petrov',19,75.5);
list:=TStringList.Create();
list.AddObject('Ivanov',p1);
list.AddObject('Petrov',p2);
(list.Objects[list.IndexOf('Ivanov')]    as    Per-
son).Show();
(list.Objects[list.IndexOf('Petrov')]    as    Per-
son).Show();
end.
```

Имеется возможность отсортировать коллекцию. Установка свойства Sorted в значение True задает автоматическую сортировку по строкам в лексикографическом порядке. Если требуется задать другой критерий сортировки, следует использовать метод CustomSort().

Как видно, набор встроенных в коллекцию TStringList методов для выполнения операций с ее элементами незначителен и основное ее назначение – предоставить удобный способ хранения пар ключ/значение.

Коллекция TList. Коллекция TList хранит список на размещенные в памяти объекты. Список представляет собой динамический массив нетипизированных (pointer) указателей, к которому можно обращаться через индексированное свойство property Items[Index:Integer]:pointer.

Работать с коллекцией TList можно, как с однонаправленным списком. Поскольку элементами коллекции являются нетипизированные указатели, она может хранить объекты любого типа, в том числе и данные встроенных типов.

Можно добавлять объекты в начало, в конец и середину коллекции. Также можно удалять объекты.

Для работы с элементами коллекции есть только два метода: сортировка по заданному критерию procedure Sort(Compare:TListSortCompare) и поиск function IndexOf(Item: Pointer): Integer.

Удобно хранить в коллекции полиморфные объекты. Наличие у них виртуальных функций позволяет, просматривая коллекцию, выполнять для каждого объекта соответствующую его классу операцию.

Следующий простой пример показывает использование коллекции TList:

```
    Var
    p1,p2:Person;
    i:integer;
    list:TList;
    begin
    p1:=Person.Create('Ivanov',25);
    p2:=Student.Create('Petrov',19,75.5);
    list:=TList.Create();
    list.Add(p1);
    list.Add(p2) ;
    for i:=0 to list.Count-1 do  Per-
son(list[i]).Show();
    end.
```

Как видно, набор встроенных в коллекцию TList методов для выполнения операций с ее элементами незначителен и основное ее назначение – предоставить удобный способ хранения объектов.

Коллекция TObjectList. Коллекция TObjectList наследует TList, но в отличие от него содержит ссылки на объекты TObject. Это позволяет более эффективно работать с полиморфными объектами.

Следующий фрагмент показывает работу с коллекцией TObjectList:

```
MyList:=TObjectList.Create();
MyList.Add(TPerson.Create('Ivanov',25));
MyList.Add(TStudent.Create('Petrov',21,50.6));
for i:=0 to Mylist.Count-1 do TPer-
son(MyList[i]).Show();
```

Коллекция TCollection. TCollection представляет собой список указателей, оптимизированный для работы с объектами определенного вида. Элемент коллекции должен быть экземпляром класса, унаследованного от TCollectionItem. Коллекция построена на основе TList, т.е. это список специального назначения.

Главное отличие TCollection от TList состоит в том, что TCollection, во первых, предназначена для хранения «визуальных» компонентов и, во-вторых, Delphi содержит средства для работы с TCollection на этапе проектирования программы, используя такие инструменты IDE Delphi, как Object Inspector и Collection Editor. Поскольку TCollection наследуется от TPersistent, Delphi умеет запоминать в файле формы все настройки коллекции и ее элементов.

7. Коллекции Java. Коллекции в языке Java объединены в библиотеке классов java.util и представляют собой контейнеры для хранения и манипулирования объектами. Коллекции Java обобщенные, и многие из методов, оперирующих коллекциями, также принимают обобщенные параметры.

Технология коллекций Java по духу подобна STL C++.

Коллекции могут сохранять только ссылки, но не примитивные значения. Для сохранения примитивных типов используется их автоматическая упаковка.

Структура коллекций Java, во-первых, позволяет различным типам коллекций работать похожим друг на друга образом и с высокой степенью способности к взаимодействию, во-вторых, она разработана в окружении набора стандартных интерфейсов, в-третьих, способна к расширению и адаптации.

В библиотеке java.util разделяются интерфейсы и реализации коллекций. Два интерфейса являются фундаментальными для коллекций. Это интерфейсы Collection и Iterator.

Интерфейс Collection<E> является корнем всей иерархии классов-коллекций. Он определяет базовую функциональность любой коллекции – набор методов, которые позволяют добавлять, удалять, выбирать элементы коллекции.

Интерфейс Iterator<E> позволяет последовательно перебрать все элементы коллекции. В Java имеется цикл «for each», который преобразуется итератором в обычный цикл с итератором. Цикл «for each» работает с любым объектом, реализующим интерфейс Iterable, а поскольку интерфейс Collection расширяет интерфейс Iterable, цикл «for each» применим для любой коллекции Java.

Итератор Java похож на итератор библиотеки STL C++. Однако между ними существует важное различие. В STL итераторы работают с коллекциями так же, как указатели с массивами. Независимо от способа обращения итератор в STL можно переместить в следующую позицию операцией i++. В Java итераторы применяются по-другому. Обращение к элементу и изменение позиции тесно связаны, поскольку единственный способ обращения к элементу основан на вызове метода next(), что приводит к перемещению на следующую позицию. В этом смысле механизм использования итераторов в Java больше похож на использование переменной POSITION для перемещения в коллекциях библиотеки MFC Visual C++.

Интерфейсы Collection и Iterator являются универсальными и для них созданы универсальные методы, которые должны реализовать классы коллекций.

Остальные интерфейсы коллекций Java:

List<E> – специализирует коллекции для обработки списков;

Set<E> – специализирует коллекции для обработки множеств, содержащих уникальные элементы.

SortedSet<E> – расширяет Set для работы с отсортированных наборов данных.

Map<K,V> – карта отображения вида «ключ-значение»;

SortedMap<K,V> – расширяет Map так, что ключи поддерживаются в возрастающем порядке.

Queue<E> – очередь.

Dequeue <E> расширяет Queue и описывает поведение двунаправленной очереди. Двунаправленная очередь может функционировать как стандартная очередь, либо как стек.

Map<K,V> – карта отображения вида «ключ-значение»;

SortedMap<K,V> – расширяет Map так, что ключи поддерживаются в возрастающем порядке.

Для работы с элементами коллекции применяются следующие интерфейсы:

Comparator<T> – для сравнения объектов;

Iterator<E>, ListIterator<E>, Map.Entry<K,V> – для перечисления и доступа к объектам коллекции.

Интерфейс Iterator<E> используется для построения объектов, которые обеспечивают доступ к элементам коллекции.

Интерфейс ListIterator<E> расширяет интерфейс Iterator<E> и предназначен в основном для работы со списками.

Интерфейс Map.Entry<K,V> предназначен для извлечения ключей и значений карты Map.

Реализации коллекций в Java

Во главе иерархии классов коллекций Java стоит абстрактный класс AbstractCollection. В этом классе оставлены абстрактными только некоторые фундаментальные методы, а остальные универсальные реализованы через них. Конкретный класс, реализующий коллекцию, может расширить класс AbstractCollection за счет реализации фундаментальных методов и при желании предложить свой вариант реализации универсальных методов интерфейса Collection.

Все классы коллекций Java можно разделить на три группы: списки, множества и карты. Во главе каждой группы стоят соответствующие абстрактные классы: AbstractList, AbstractSet и AbstractMap. Эти классы реализуют основную функциональность, определенную в интерфейсах List, Set, Map. Класс AbstractList имеет потомок AbstractSequentialList. Основное его отличие от AbstractList заключается в том, что этот класс обеспечивает не только последовательный, но и произвольный доступ к элементам списка.

Карта отображений Map – это объект, который хранит пару «ключ-значение». Поиск объекта (значения) облегчается по сравнению с множествами за счет того, что его можно найти по его уникальному ключу. Уникальность объектов-ключей должна обеспечиваться переопределением методов hashCode() и equals() пользовательским классом.

Если элемент с указанным ключом отсутствует в карте, то возвращается значение null.

Конкретные классы коллекций создаются как экземпляры следующих классов:

– класс ArrayList<E> реализует динамический массив объектных ссылок. Расширяет класс AbstractList<E> и реализует интерфейс List<E>. Удаление и добавление элементов для такой коллекции представляет собой ресурсоемкую задачу, поэтому объект ArrayList<E> лучше всего подходит для хранения неизменяемых списков;

– класс LinkedList<E> реализует связанный список. Класс расширяет AbstractSequentalList и реализует интерфейсы List, Dequeue и Queue. В отличие от массива, который хранит объекты в последовательных ячейках памяти, связанный список хранит объекты отдельно, но вместе со ссылками на следующие и предыдущие элементы последовательности. LinkedList является двухсвязным списком и позволяет перемещаться как от начала в конец списка, так и наоборот. LinkedList удобно использовать для организации очереди и стека;

– класс ArrayDeque<E> расширяет AbstractCollection и реализует интерфейс Deque;

– класс HashSet<E> наследуется от AbstractSet<E> и реализует интерфейс Set<E>, используя хэш-таблицу для хранения коллекции. Ключ (хэш-код) используется вместо индекса для доступа к данным, что значительно ускоряет поиск определенного элемента;

– класс TreeSet<E> для хранения объектов использует бинарное дерево. При добавлении объекта в дерево он сразу же размещается в необходимую позицию с учетом сортировки. Сортировка происходит благодаря тому, что все добавляемые элементы реализуют интерфейсы Comparator и Comparable. Обработка операций удаления и вставки объектов происходит медленнее, чем в хэш-множествах, но быстрее, чем в списках;

– класс HashMap<K,V> расширяет класс AbstractMap<K,V> и реализует интефейс Map. Этот класс использует хэш-таблицу, в которой ключи отсортированы относительно значений их хэш-кодов;

– класс TreeMap<K,V> расширяет AbstractMap<K,V>, используя дерево, где ключи расположены в виде дерева поиска в отсортированном порядке.

Для работы с коллекциями в Java определен класс Collections. Класс Collections является классом-утилитой и содержит несколько вспомогательных методов для работы с классами, обеспечивающими различные интерфейсы коллекций. Например, для сортировки элементов списков, поиска элементов в упорядоченных коллекциях и т.д.

Все методы класса Collections статические, ими можно пользоваться, не создавая экземпляры классу Collections. Эти методы играют ту же роль, что алгоритмы в STL C++. Как обычно, в статических методах коллекция, с которой работает метод, задается его аргументом.

Для того чтобы гарантировать, что в коллекцию не будет помещен «посторонний» объект, в класс Collections добавлен метод – checkedCollection():

```
public static <E> Collection <E>
checkedCollection(Collection<E> c, Class<E> type)
```

Этот метод создает коллекцию, проверяемую на этапе выполнения, то есть в случае добавления «постороннего» объекта генерируется исключение ClassCastException.

Например, создадим коллекцию ArrayList для хранения строк:

```
Collection c=  new ArrayList<String>();
```

Коллекция позволит добавить объект, не являющийся строкой, например:

```
 c.add("Java");  c.add("Swing"); c.add(77);
```

Если создать коллекцию методом checkedCollection:

```
Collection c=Collections.checkedCollection(new
ArrayList<String>(), String.class),
```

попытка добавить целое число приведет к исключительной ситуации.

В этот же класс добавлен целый ряд методов, специализированных для проверки конкретных типов коллекций, а именно: `checkedList()`, `checkedSortedMap()`, `checkedMap()`, `checkedSortedSet()`, `checkedSet()`.

8. Коллекции C#. В C# имеется четыре типа коллекций: необобщенные, специализированные, битовые и обобщенные. Рассмотрим три из них. Необобщенные коллекции и их интерфейсы определены в пространстве имен System.Collections.

Необобщенные коллекции реализуют основные структуры данных, такие как динамический массив, стек, очередь, карта. Необобщенные коллекции оперируют ссылками типа object, поэтому их можно использовать для хранения объектов любого типа, причем в одной

коллекции можно хранить объекты разных типов. С одной стороны, это удобно, но с другой – не обеспечивает типовую безопасность.

Специализированные коллекции определены в пространстве имен System.Collections.Specialized и оптимизированы для работы с определенными типами данных или для работы особым образом.

Обобщенные коллекции и их интерфейсы определены в пространстве имен System.Collections.Generic. Эти коллекции основаны на шаблонах («обобщениях» в терминологии C#) и хранят только такие элементы, которые совместимы с их параметрами шаблона (аргументами типа). Таким образом, обобщенные коллекции обеспечивают типовую безопасность.

Во многих случаях обобщенные коллекции – это просто обобщенные эквиваленты необобщенных коллекций, однако есть и обобщенные коллекции, не имеющие эквивалента среди необобщенных.

Так же, как и в Java, в C# коллекции аналогичны контейнерам STL C++.

Интерфейсы коллекций C#. Все коллекции C# реализуют так называемые нумераторы, которые поддерживают интерфейсы IEnumerator и IEnumerable. Нумератор обеспечивает стандартный последовательный доступ к элементам коллекции. Каждая коллекция должна реализовать интерфейс IEnumerable для возможности доступа к ее элементам с помощью методов интерфейса IEnumerator. Нумератор используется также для прохода по коллекции с помощью цикла foreach.

Интерфейс ICollection/ICollection<T> наследуется от IEnumerable/ IEnumerable<T> и является базовым интерфейсом всех коллекций. В нем объявлены методы и свойства, которые имеют все коллекции.

Интерфейс IList/IList<T> наследуется от ICollection/ICollection<T> и определяет поведение коллекции, к элементам которой возможен доступ с помощью индекса.

Интерфейс IComparer/ IComparer<T> определяет метод Compare(), который задает способ сравнения двух объектов.

Интерфейс IDictionary/ IDictionary<TK,TV> определяют коллекцию, которая хранит пару ключ/ значение.

Реализации необобщенных коллекций. ArrayList – динамический массив. Имеются методы для поиска и сортировки массива. Позволяет обращаться к элементу коллекции по индексу. При выходе

индекса за границу генерирует исключение. Hahtable – хэш-таблица для пар ключ/значение. Хеш-таблица не гарантирует упорядочивание ее элементов. Queue – очередь. SortedList – сортированный список пар ключ/значение. Определяет коллекцию, которая хранит пары ключ/значение в отсортированном виде в соответствии со значениями ключа. Stack – стек.

Реализации обобщенных коллекций. List<T> – динамический массив. Обеспечивает функциональность, аналогичную функциональности необобщенного класса ArrayList. LinkedList<T> – обобщенной двусвязный список. Не имеет аналога среди необобщенных коллекций. Подобно большинству реализаций двусвязного списка класс LinkedList<T> хранит значения своих элементов в узлах, имеющих ссылки на предыдущий и следующий элементы списка. Эти узлы являются объектами типа LinkedListNode<T>. Возможность перемещения в обоих направлениях особенно важна в таких приложениях, как базы данных. Dictionary<TK,TV> – словарь. Хранит пары ключ/значение. Обеспечивает функциональность, аналогичную функциональности необобщенной коллекции Hahtable. SortedDictionary<TK,TV> хранит пары ключ/значение. Аналогичен классу Dictionary<TK,TV>, за исключением того, что его элементы отсортированы по значениям ключей. В классе определен индексатор, который позволяет получать или задавать значение элемента, используя в качестве индекса ключ. Его также можно использовать для добавления нового элемента. По функциональности аналогичен карте Map библиотеки STL C++. SortedList<TK,TV> хранит отсортированный список пар ключ/значение. Аналогичен классу SortedDictionary<TK,TV>, но имеет другие характеристики времени выполнения. Queue<T> – обобщенный эквивалент необобщенного класса Queue. Stack<T> – обобщенный эквивалент необобщенного класса Stack.

Для работы с коллекциями в их классах определено множество методов, которые обеспечивают большинство широко используемых алгоритмов. Эти методы по своей функциональности аналогичны алгоритмам STL C++.

Выводы

1. В качестве простейших структур хранения наборов данных можно использовать обычные массивы. Однако функциональность использования массивов невелика, к тому же не все языки программирования позволяют контролировать выход индекса за границы.

2. Наиболее эффективно использование массивов для хранения наборов данных в языке C# за счет представления массивов как объектов специального класса Array.

3. Библиотеки классов коллекций STL C++, Java и C# предоставляют примерно одинаковую функциональность. Они предоставляют стандартный набор всех основных структур хранения данных: динамические массивы, списки, стеки, очереди, ассоциативные массивы. Функциональность коллекций Delphi значительно ниже.

4. Для работы с коллекциями в STL C++ и Java алгоритмы определены вне классов коллекций (в STL C++ как обобщенные глобальные функции, в Java как статические методы специального класса), что, на наш взгляд, позволяет более эффективно их использовать.

5. Если вы предполагаете хранить в коллекциях элементы одного типа, то лучше использовать обобщенные коллекции как обеспечивающие типовую безопасность.

6. Если необходимо хранить в коллекциях элементы разных типов, то приходится использовать необобщенные коллекции. Однако они не обеспечивают типовую безопасность.

Получено 27.09.2010