# ГЛАВА 10. СТРУКТУРЫ ДАННЫХ

## Оглавление



## §10.1 Шаблоны функций

**Шаблоны функций** – это инструментарии, согласно которым создаются локальные версии шаблонированной функции для определенного набора параметров и типов данных.

Данные инструменты упрощают труд программиста. Например, нам нужно запрограммировать функцию, которая выводила бы на экран элементы массива. Чтобы написать такую функцию, мы должны знать тип данных массива, который будем выводить на экран. И тут возникает трудность в том, что тип данных не один, мы хотим, чтобы функция выводила массивы типа `int`, `double`, `float` и `char`.

Необходимо запрограммировать целых 4 функции, которые выполняют одни и те же действия, но для различных типов данных. Так как мы еще не знакомы с шаблонами функций, мы поступим так: воспользуемся перегрузкой функций.

**Листинг 11.1**

```
1  void print(const int *a, int c)
2  {   for (int i = 0; i < c; i++)
3  cout << a[i] << "   ";  }
4  void print(const double *a, int c)
5  {   for (int i = 0; i < c; i++)
6  cout << a[i] << "   ";  }
7  void print(const float *a, int c)
8  {   for (int i = 0; i < c; i++)
9  cout << a[i] << "   ";  }
10 void print(const char *a, int c)
11 {   for (int i = 0; i < c; i++)
12 cout << a[i] << "   ";  }
```

Таким образом, мы имеем 4 перегруженные функции, для разных типов данных. Они отличаются только заголовком функции, тело у них абсолютно одинаковое.

Создадим один шаблон, в котором описываем все типы данных. Исходный код не будет захламляться никому ненужными строками кода. Пример программы с шаблоном функции приведен ниже.

**Листинг 11.2**

```
1  #include <iostream>
2  #include <cstring>
3  using namespace std;
4  template <typename T>
5  void printArray(const T * array, int count)
6  {
7      for (int ix = 0; ix < count; ix++)
8          cout << array[ix] << "   ";
9      cout << endl;
10 }
11 int main()
12 {
13 const int iSize = 10,
```

```
14  int iArray[iSize] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
15  cout << "\nМассив типа int:\n";
16  printArray(iArray, iSize); }
```

Все шаблоны функций начинаются со слова **`template`**, после которого идут угловые скобки, в которых перечисляется список параметров. Каждому параметру должно предшествовать зарезервированное слово `class` или `typename`.

Ключевое слово `typename` говорит о том, что в шаблоне будет использоваться встроенный тип данных, такой как: int, double, float, char и т. д. А ключевое слово class сообщает компилятору, что в шаблоне функции в качестве параметра будут использоваться пользовательские типы данных, то есть классы.

У нас в шаблоне функции использовались встроенные типы данных, поэтому в строке 4 мы написали `template <typename T>`. Вместо `T` можно подставить любое другое имя, какое только придумаете.

## §10.2 Контейнерные классы

**Контейнерный класс** (или ещё **«класс-контейнер»**) в C++ — это **класс**, предназначенный для хранения и обработки нескольких объектов определённого типа данных (пользовательских или встроенных). Существует много разных контейнерных классов, каждый из которых имеет свои преимущества, недостатки или ограничения в использовании. Безусловно, наиболее часто используемым контейнером в программировании является **массив**. Хотя в C++ есть стандартные обычные массивы, большинство программистов используют контейнерные классы-массивы: **std::array** или **std::vector** из-за преимуществ, которые они предоставляют. В отличие от стандартных массивов, контейнерные классы-массивы имеют возможность динамического изменения своего размера, когда элементы добавляются или удаляются. Это не только делает их более удобными, чем обычные массивы, но и безопаснее.

Обычно, **функционал классов-контейнеров** в C++ следующий:

1. Создание пустого контейнера (через **конструктор**).
2. Добавление нового объекта в контейнер.
3. Удаление объекта из контейнера.
4. Просмотр количества объектов, находящихся на данный момент в контейнере.
5. Очистка контейнера от всех объектов.
6. Доступ к сохранённым объектам.
7. Сортировка объектов/элементов (опционально).

В отличие от реальной жизни, когда контейнеры могут хранить любые типы объектов, которые в них помещают, в C++ контейнеры обычно содержат только один тип данных. Например, если у вас целочисленный массив, то он может содержать только целочисленные значения. В отличие от некоторых других языков, C++ не позволяет смешивать разные типы данных внутри контейнеров. Если вам нужны контейнеры для хранения значений типов int и double, то вам придётся написать два отдельных контейнера (или использовать шаблоны, о которых мы будем говорить в следующих параграфах).

## §10.3 Контейнерный класс-массив

Сейчас мы напишем целочисленный класс-массив с нуля, реализуя функционал контейнеров в C++. Этот класс-массив будет типа контейнера значения, в котором будут храниться копии элементов, которые он содержит.

Наш класс `ArrayInt` должен отслеживать два значения: непосредственно данные и свою длину. Поскольку мы хотим, чтобы наш массив мог изменять свою длину, то нам нужно использовать динамическое выделение памяти, что означает, что мы будем использовать указатель для хранения данных:

**Листинг 11.3**

```
1  class ArrayInt
2  {
3  private:
4      int m_length;
5      int *m_data;
```

```
6  };
```

Теперь нам нужно добавить конструкторы, чтобы иметь возможность создавать объекты класса `ArrayInt`. Мы добавим два конструктора: первый будет создавать пустой массив, второй — массив заданного размера:

**Листинг 11.4**

```
1  class ArrayInt
2  {
3  private:
4      int m_length;
5      int *m_data;
6  public:
7      ArrayInt(): m_length(0), m_data(NULL)
8      {
9      }
10     ArrayInt(int length): m_length(length)
11     {
12         assert(length >= 0);
13         if (length > 0)
14             m_data = new int[length];
15         else
16             m_data = NULL;
17     }
18 };
```

Нам также потребуются функции, которые будут выполнять очистку `ArrayInt`. Во-первых, добавим деструктор, который будет просто освобождать любую динамически выделенную память. Во-вторых, напишем функцию `erase()`, которая будет выполнять очистку массива и сбрасывать его длину на 0:

**Листинг 11.5**

```
1  ~ArrayInt()
2  {
3  delete[] m_data;
4  // Здесь нам не нужно присваивать значение NULL для m_data или выполнять m_length = 0, так как объект и так будет уничтожен
5  }
6  void erase()
7  {
```

```
8   delete[] m_data;
9   // Здесь нам нужно указать m_data значение NULL, чтобы на выходе не было висячего указателя
10  m_data = NULL;
11  m_length = 0;
12  }
```

Теперь **перегрузим оператор индексации []**, чтобы иметь доступ к элементам массива. Мы также должны выполнить проверку корректности передаваемого индекса, что лучше всего сделать с помощью **выражения assert()**. Также добавим **функцию доступа** для возврата длины массива:

**Листинг 11.6**

```
1   class ArrayInt
2   {
3   private:
4       int m_length;
5       int *m_data;
6   public:
7       ArrayInt():
8           m_length(0), m_data(NULL)
9       {
10      }
11      ArrayInt(int length):
12          m_length(length)
13      {
14          assert(length >= 0);
15          if (length > 0)
16              m_data = new int[length];
17          else
18              m_data = NULL;
19      }
20      ~ArrayInt()
21      {
22          delete[] m_data;
23      }
24      void erase()
25      {
26          delete[] m_data;
27          // Указываем m_data значение NULL, чтобы на выходе не было висячего указателя
28          m_data = NULL;
```

```
29          m_length = 0;
30      }
31      int& operator[](int index)
32      {
33          assert(index >= 0 && index < m_length);
34          return m_data[index];
35      }
36      int getLength() { return m_length; }
37  };
```

Теперь у нас уже есть класс `ArrayInt`, который мы можем использовать. Мы можем выделить массив определённого размера и использовать оператор [] для извлечения или изменения значений элементов.

Тем не менее, есть ещё несколько вещей, которые мы не можем выполнить с нашим `ArrayInt`. Это автоматическое изменение размера массива, добавление/удаление элементов, сортировка элементов.

Во-первых, давайте реализуем возможность массива изменять свой размер. Напишем две разные функции для этого. Первая функция: `reallocate()`, при изменении размера массива будет уничтожать все существующие элементы — это быстро. Вторая функция: `resize()`, при изменении размера массива будет сохранять все существующие элементы — это медленно.

**Листинг 11.7**

```
1   void reallocate(int newLength)
2   {
3   // Удаляем все существующие элементы внутри массива
4   erase();
5   // Если наш массив должен быть пустым, то выполняем
    возврат здесь
6   if (newLength <= 0)
7   return;
8   // Дальше нам нужно выделить новые элементы
9   m_data = new int[newLength];
10  m_length = newLength;
11  }
12
13  void resize(int newLength)
```

```
14  {
15  // Если массив уже нужной длины, то выполняем return
16  if (newLength == m_length)
17  return;
18  // Если нужно сделать массив пустым, то делаем это и затем выполняем return
19  if (newLength <= 0)
20  {
21  erase();
22  return;
23  }
24  // Теперь мы можем предположить, что newLength состоит хотя бы из одного элемента. Выполняется следующий алгоритм:
25  // 1. Выделяем  память под новый массив.
26  // 2. Копируем элементы из существующего массива в наш, только что выделенный, массив.
27  // 3. Уничтожаем старый массив и присваиваем m_data адрес нового массива.
28  // Выделяем новый массив
29  int *data = new int[newLength];
30  // Затем нам нужно разобраться с количеством копируемых элементов в новый массив
31  // Нам нужно скопировать столько элементов, сколько их есть в меньшем из массивов
32  if (m_length > 0)
33  {
34  int elementsToCopy = (newLength > m_length) ? m_length : newLength;
35  // Поочерёдно копируем элементы
36  for (int index=0; index < elementsToCopy; ++index)
37  data[index] = m_data[index];
38  }
39  // Удаляем старый массив, так как он нам уже не нужен
40  delete[] m_data;
41  // И используем вместо старого массива новый! Обратите внимание, m_data указывает на тот же адрес, на который указывает наш новый динамически выделенный массив.
42  // Поскольку данные были динамически выделенные, то они не будут уничтожены, когда выйдут из области видимости
43  m_data = data;
44  m_length = newLength;
45  }
```

Реализуем возможность добавления/удаления элементов из контейнера.

**Листинг 11.8**

```
1  void insertBefore(int value, int index)
2      {
3          // Создаём новый массив на один элемент больше старого массива
4          int *data = new int[m_length+1];
5          // Копируем все элементы до index-а добавляемого элемента
6          for (int before=0; before < index; ++before)
7              data[before] = m_data[before];
8          // Вставляем наш новый элемент в наш новый массив
9          data [index] = value;
10         // Копируем все значения после вставляемого элемента
11         for (int after=index; after < m_length; ++after)
12             data[after+1] = m_data[after];
13         // Удаляем старый массив и используем вместо него новый массив
14         delete[] m_data;
15         m_data = data;
16         ++m_length;
17     }
18     void remove(int index)
19     {
20         // Если это последний элемент массива, то делаем массив пустым и выполняем return
21         if (m_length == 1)
22         {
23             erase();
24             return;
25         }
26         // Создаём новый массив на один элемент меньше нашего старого массива
27         int *data = new int[m_length-1];
28         // Копируем все элементы до index-а удаляемого элемента
29         for (int before=0; before < index; ++before)
30             data[before] = m_data[before];
```

```
31          // Копируем все значения после удаляемого элемента
32          for (int after=index+1; after < m_length; ++after )
33              data[after-1] = m_data[after];
34          // Удаляем старый массив и используем вместо него новый массив
35          delete[] m_data;
36          m_data = data;
37          --m_length;
38      }
39      // Несколько дополнительных функций просто для удобства
40      void insertAtBeginning(int value) { insertBefore(value, 0); }
41      void insertAtEnd(int value) { insertBefore(value, m_length); }
```

Хотя написание контейнерных классов может быть несколько сложным, но хорошая новость заключается в том, что вам их нужно написать только один раз. Как только контейнерный класс работает, вы можете его повторно использовать где-угодно без каких-либо дополнительных действий/усилий по части программирования.

Также стоит отметить, что, хотя наш контейнерный класс `ArrayInt` содержит фундаментальный тип данных (int), мы также могли бы легко использовать и пользовательский тип данных.

## §10.4 Шаблоны классов

Шаблон класса позволяет задать тип для объектов, используемых в классе.

Из урока о контейнерных классах мы узнали, как, используя композицию, реализовать классы, содержащие несколько объектов определённого типа данных. В качестве примера мы использовали класс `ArrayInt`.

Хотя этот класс обеспечивает простой способ создания массива целочисленных значений, что, если нам нужно будет работать со значениями

типа double? Используя традиционные методы программирования, мы бы создали новый класс ArrayDouble для работы со значениями типа double.

Хотя кода много, но классы почти идентичны, меняется только тип данных! Как вы уже могли бы догадаться, это идеальный случай для использования шаблонов. Создание шаблона класса аналогично созданию шаблона функции. Например, создадим шаблон класса Array:

**Листинг 11.9**

```
1  template <class T> // это шаблон класса с T вместо передаваемого типа данных
2  class Array
3  {
4  private:
5      int m_length;
6      T *m_data;
7  public:
8      Array()
9      {
10         m_length = 0;
11         m_data = NULL;
12     }
13     Array(int length)
14     {
15         m_data = new T[length];
16         m_length = length;
17     }
18     ~Array()
19     {
20         delete[] m_data;
21     }
22     void Erase()
23     {
24         delete[] m_data;
25         // Присваиваем значение NULL для m_data, чтобы на выходе не получить висячий указатель!
26         m_data = NULL;
27         m_length = 0;
28     }
29     T& operator[](int index)
30     {
```

```
31          return m_data[index];
32      }
33      int getLength(); // определяем метод и шаблон метода getLength() ниже
34  };
35  template <typename T> // метод, определённый вне тела класса, нуждается в собственном определении шаблона метода
36  int Array<T>::getLength() { return m_length; } // обратите внимание, имя класса - Array<T>, а не просто Array
```

Как вы можете видеть, эта версия почти идентична версии `ArrayInt`, за исключением того, что мы добавили объявление параметра шаблона класса и изменили тип данных с int-а на `T`.

Вот пример использования шаблона класса `Array`:

**Листинг 11.10**

```
1   #include <iostream>
2   #include "Array.h"
3   int main()
4   {
5   Array<int> intArray(10);
6   Array<double> doubleArray(10);
7   for (int count = 0; count < intArray.getLength(); ++count)
8   {
9   intArray[count] = count;
10  doubleArray[count] = count + 0.5;
11  }
12  for (int count = intArray.getLength()-1; count >= 0; --count)
13  std::cout << intArray[count] << "\t" << doubleArray[count] << '\n';
14  return 0;
15  }
```

Шаблоны классов работают точно так же, как и шаблоны функций: компилятор копирует шаблон класса, заменяя типы параметров шаблона класса на передаваемые типы данных, а затем компилирует эту копию. Если у

вас есть шаблон класса, но вы его не используете, то компилятор не будет его даже компилировать.

Шаблоны классов идеально подходят для реализации контейнерных классов, так как очень часто таким классам приходится работать с разными типами данных, а шаблоны позволяют это организовать в минимальном количестве кода.

## §10.5 Обобщенные типы

Кроме обычных типов фреймворк .NET также поддерживает обобщенные типы (generics), а также создание обобщенных методов. Чтобы разобраться в особенности данного явления, сначала посмотрим на проблему, которая могла возникнуть до появления обобщенных типов. Посмотрим на примере. Допустим, мы определяем класс для представления банковского счета. К примеру, он мог бы выглядеть следующим образом:

**Листинг 11.11**

```
1  class Account
2  {
3      public int Id { get; set; }
4      public int Sum { get; set; }
5  }
```

Класс `Account` определяет два свойства: `Id` - уникальный идентификатор и `Sum` - сумму на счете.

Здесь идентификатор задан как числовое значение, то есть банковские счета будут иметь значения 1, 2, 3, 4 и так далее. Однако также нередко для идентификатора используются и строковые значения. И у числовых, и у строковых значений есть свои плюсы и минусы. И на момент написания класса мы можем точно не знать, что лучше выбрать для хранения идентификатора - строки или числа. Либо, возможно, этот класс будет использоваться другими разработчиками, которые могут иметь свое мнение по данной проблеме.

И на первый взгляд, чтобы выйти из подобной ситуации, мы можем определить свойство `Id` как свойство типа `object`. Так как тип `object` является универсальным типом, от которого наследуется все типы,

соответственно в свойствах подобного типа мы можем сохранить и строки, и числа:

**Листинг 11.12**

```
1  class Account
2  {
3      public object Id { get; set; }
4      public int Sum { get; set; }
5  }
```

Затем этот класс можно было использовать для создания банковских счетов в программе:

**Листинг 11.13**

```
1  Account account1 = new Account { Sum = 5000 };
2  Account account2 = new Account { Sum = 4000 };
3  account1.Id = 2;
4  account2.Id = "4356";
5  int id1 = (int)account1.Id;
6  string id2 = (string)account2.Id;
7  Console.WriteLine(id1);
8  Console.WriteLine(id2);
```

Все вроде замечательно работает, но такое решение является не очень оптимальным. Дело в том, что в данном случае мы сталкиваемся с такими явлениями как **упаковка** (boxing) и **распаковка** (unboxing).

Так, при присвоении свойству `Id` значения типа int, происходит упаковка этого значения в тип `Object`:

```
account1.Id = 2; // упаковка значения int в тип Object
```

Чтобы обратно получить данные в переменную типов int, необходимо выполнить распаковку:

```
int id1 = (int)account1.Id; // Распаковка в тип int
```

Упаковка (boxing) предполагает преобразование объекта значимого типа (например, типа int) к типу object. При упаковке общеязыковая среда CLR обертывает значение в объект типа System.Object и сохраняет его в управляемой куче (хипе). Распаковка (unboxing), наоборот, предполагает преобразование объекта типа object к значимому типу. Упаковка и распаковка

ведут к снижению производительности, так как системе надо осуществить необходимые преобразования.

Кроме того, существует другая проблема - проблема безопасности типов. Так, мы получим ошибку во время выполнения программы, если напишем следующим образом:

**Листинг 11.14**

```
1  Account account2 = new Account { Sum = 4000 };
2  account2.Id = "4356";
3  int id2 = (int)account2.Id; // Исключение InvalidCastException
```

Мы можем не знать, какой именно объект представляет `Id`, и при попытке получить число в данном случае мы столкнемся с исключением InvalidCastException.

Эти проблемы были призваны устранить обобщенные типы (также часто называют универсальными типами). Обобщенные типы позволяют указать конкретный тип, который будет использоваться. Поэтому определим класс `Account` как обобщенный:

**Листинг 11.15**

```
1  class Account<T>
2  {
3      public T Id { get; set; }
4      public int Sum { get; set; }
5  }
```

Угловые скобки в описании `class Account<T>` указывают, что класс является обобщенным, а тип `T`, заключенный в угловые скобки, будет использоваться этим классом. Необязательно использовать именно букву `T`, это может быть и любая другая буква или набор символов. Причем сейчас нам неизвестно, что это будет за тип, это может быть любой тип. Поэтому параметр `T` в угловых скобках еще называется универсальным параметром, так как вместо него можно подставить любой тип.

**Листинг 11.16**

```
1  Account<int> account1 = new Account<int> { Sum = 5000 }
```

```
2  Account<string> account2 = new Account<string> { Sum = 4000 }
3  account1.Id = 2 // упаковка не нужна
4  account2.Id = "4356"
5  int id1 = account1.Id // распаковка не нужна
6  string id2 = account2.Id
7  Console.WriteLine(id1)
8  Console.WriteLine(id2)
```

Поскольку класс `Account` является обобщенным, то при определении переменной после названия типа в угловых скобках необходимо указать тот тип, который будет использоваться вместо универсального параметра `T`. В данном случае объекты `Account` типизируется типами int и string.

Поэтому у первого объекта `account1` свойство `Id` будет иметь тип int, а у объекта `account2` - тип string.

При попытке присвоить значение свойства `Id` переменной другого типа мы получим ошибку компиляции.

Тем самым мы избежим проблем с безопасностью типов. Таким образом, используя обобщенный вариант класса, мы снижаем время на выполнение и количество потенциальных ошибок.

Иногда возникает необходимость присвоить переменным универсальных параметров некоторое начальное значение, в том числе и `null`.

В этом случае нам надо использовать оператор `default(T)`. Он присваивает ссылочным типам в качестве значения `null`, а типам значений - значение 0:

**Листинг 11.17**

```
1  class Account<T>
2  {
3      T id = default(T);
4  }
```

Обобщения могут использовать несколько универсальных параметров одновременно, которые могут представлять различные типы:

**Листинг 11.18**

```
1  class Transaction<U, V>
```

```
2  {
3      public U FromAccount { get; set; } // с какого счета перевод
4      public U ToAccount { get; set; } // на какой счет перевод
5      public V Code { get; set; } // код операции
6      public int Sum { get; set; } // сумма перевода
7  }
```

Здесь класс `Transaction` использует два универсальных параметра. Применим данный класс:

**Листинг 11.19**

```
1  Account<int> acc1 = new Account<int> { Id = 1857, Sum = 4500 };
2  Account<int> acc2 = new Account<int> { Id = 3453, Sum = 5000 };
3  Transaction<Account<int>, string> transaction1 = new Transaction<Account<int>, string>
4  {
5      FromAccount = acc1,
6      ToAccount = acc2,
7      Code = "45478758",
8      Sum = 900
9  };
```

Здесь объект `Transaction` типизируется типами `Account<int>` и `string`. То есть в качестве универсального параметра `U` используется класс `Account<int>`, а для параметра `V` - тип `string`. При этом, как можно заметить, класс, которым типизируется `Transaction`, сам является обобщенным.

Кроме обобщенных классов можно также создавать обобщенные методы, которые точно также будут использовать универсальные параметры. Например:

**Листинг 11.20**

```
1  class Program
2  {
3      private static void Main(string[] args)
4      {
5          int x = 7;
```

```
6           int y = 25;
7           Swap<int>(ref x, ref y); // или так Swap(ref x, ref y);
8           Console.WriteLine($"x={x}   y={y}");  // x=25 y=7
9           string s1 = "hello";
10          string s2 = "bye";
11          Swap<string>(ref s1, ref s2); // или так Swap(ref s1, ref s2);
12          Console.WriteLine($"s1={s1}     s2={s2}"); // s1=bye   s2=hello
13          Console.Read();
14      }
15      public static void Swap<T> (ref T x, ref T y)
16      {
17          T temp = x;
18          x = y;
19          y = temp;
20      }
21  }
```

Здесь определен обобщенный метод `Swap`, который принимает параметры по ссылке и меняет их значения. При этом в данном случае не важно, какой тип представляют эти параметры.

В методе `Main` вызываем метод `Swap`, типизируем его определенным типом и передаем ему некоторые значения.

## §10.6 Стандартная библиотека шаблонов

**STL** (Standard Template Library) – стандартная библиотека шаблонов, включённая в новые версии стандарта C++. STL обеспечивает стандартные классы и функции, которые реализуют наиболее популярные и широко используемые алгоритмы и структуры данных. STL строится на основе шаблонов классов, поэтому входящие в неё алгоритмы и структуры применимы почти ко всем типам данных. Ядро библиотеки образуют три элемента: контейнеры, алгоритмы и итераторы.

**Контейнеры** (containers) - это объекты, предназначенные для хранения набора элементов. Например, вектор, линейный список, множество. В каждом классе-контейнере определен набор функций для работы с ними. Например, контейнер «список» содержит функции для вставки, удаления и слияния элементов.

**Алгоритмы** (algorithms) выполняют операции над содержимым контейнера. Существуют алгоритмы для инициализации, сортировки, поиска, замены содержимого контейнеров.

**Итераторы** (iterators) - это объекты, которые по отношению к контейнеру играют роль указателей. Они позволяют получить доступ к содержимому контейнера и сканировать его элементы, примерно так же, как указатели используются для доступа к элементам массива. С итераторами можно работать так же, как с указателями. К ним можно применять операции *, инкремент, декремент.

### §10.6.1 Контейнеры STL

Безусловно, наиболее часто используемым функционалом **STL** являются **контейнерные классы** (или как их ещё называют — «**контейнеры**»). STL содержит много разных контейнерных классов, которые можно использовать в разных ситуациях. Если говорить в общем, то **контейнеры STL делятся на две основные категории**:

1. последовательные контейнеры;
2. ассоциативные контейнеры.

#### §10.6.1.1 Последовательные контейнеры

**Последовательные контейнеры** (или ещё «**контейнеры последовательности**») — это контейнерные классы, элементы которых находятся в последовательности. Их определяющей характеристикой является то, что вы можете вставить свой элемент в любое место контейнера. Наиболее распространённым примером последовательного контейнера является **массив**: при вставке 4-ёх элементов в массив, эти элементы будут находиться (в массиве) в точно таком же порядке, в котором вы их вставляли.

Ниже приведены последовательные контейнеры библиотеки STL:

1. `vector`. Работает как динамический массив и увеличивается с конца. Вектор похож на книжную полку, книги можно добавлять или удалять по одной с конца.
2. `deque`. Подобен контейнеру `vector`, но новые элементы можно вставлять и удалять также вначале.
3. `list`. Работает как двусвязный список. Список похож на цепь, где каждый объект связан со следующим звеном. Вы можете добавить или удалить звено (т.е. объект) в любой позиции.
4. `forward_list`. Подобен списку `list`, но односвязный список позволяет осуществить перебор только в одном направлении.

Класс **vector** (или просто «вектор») — это динамический массив, способный увеличиваться по мере необходимости для содержания всех своих

элементов. Для создания вектора нам понадобится подключить библиотеку — <vector>, в ней хранится шаблон вектора.

Объявление вектора происходит благодаря использованию следующей конструкции:

```
1 vector <тип данных> имя_вектора;
```

Вектор можно проинициализировать начальным набором данных при его создании:

```
1 vector <int> vec1 = {2, 4, 6};
```

Для доступа к элементам вектора можно использовать квадратные скобки [], также, как и для обычных массивов. Но в C++ есть еще один способ это сделать благодаря функции — `at()`. В скобках мы должны указать индекс той ячейки, к которой нужно обратиться. Пример обращения к элементам вектора имеет следующий вид:

```
1 cout << vec1.at(1); // = 4
```

С помощью функции insert() можно добавлять элементы в начало вектора. Эта функция имеет такую конструкцию:

```
1 <имя вектора>.insert(<итератор>, <значение>);
```

Указывать размер вектора можно по-разному. Можно это сделать еще при его инициализации, а можно хоть в самом конце программы. Вот, например, способ указать длину вектора на старте:

```
1 vector <int> vector_first(5);
```

Так в круглых скобках () после имени вектора указываем первоначальную длину. А вот второй способ:

```
1 vector <int> vector_second;  // создали вектор
2 vector_second.reserve(5);  // указали число ячеек
```

Понятно, что вам может понадобиться записать числа в двумерный массив. Но зачем использовать массив, если можно оперировать векторами.

```
1 vector < vector < тип данных > > имя_вектора;
```

Как можно увидеть, нам пришлось только добавить слова `vector` и еще его `<тип>`.

А чтобы указать количества векторов в векторе нам потребуется — метод `resize()`.

```
1  vector < vector <int> > vec;
2  vec.resize(10);  // десять векторов
```

Если нам требуется узнать длину вектора, понадобится функция — `size()`. Эта функция практически всегда используется вместе с циклом `for`.

```
1  for (int i = 0; i < vector.size(); i++) {
2    // ...
3  }
```

Также, если нам требуется узнать пуст ли стек, мы можем использовать функцию — `empty()`. При отсутствии в ячейках какого-либо значения — это функция возвратит — `true`. В противном случае результатом будет — `false`.

С помощью функции `push_back()` мы можем добавить ячейку в конец вектора, а функция `pop_back()` все делает наоборот — удаляет одну ячейку в конце вектора. Использование функции `push_back()` без указания значения ячейки — невозможно.

Метод `clear()` используется для удаления всех элементов вектора.

В следующей программе мы вставляем 5 целых чисел в вектор и с помощью перегруженного оператора индексации `[]` получаем к ним доступ для их последующего вывода:

**Листинг 11.21**

```
1  #include <iostream>
2  #include <vector>
3  using namespace std;
4  int main()
5  {
6  vector<int> vect;
7  for (int count=0; count < 5; ++count)
8  vect.push_back(10 - count); // вставляем числа в конец массива
9  for (int index=0; index < vect.size(); ++index)
10 cout << vect[index] << ' ';
11 cout << '\n';
12 }
```

Результат выполнения программы выше:

**10 9 8 7 6**

Класс **deque** (или просто «**дек**») — это двусторонняя очередь, реализованная в виде динамического массива, который может расти с обоих концов. Контейнер похож на `vector`, но с возможностью быстрой вставки и удаления элементов на обоих концах за $O$ (1).

Для добавления элемента в начало двусторонней очереди используется метод `push_front()`, для удаления элемента используется метод `pop_front()`.

**Листинг 11.22**

```
1  #include <iostream>
2  #include <deque>
3  int main()
4  {
5  deque<int> deq; // инициализация очереди
6  for (int count=0; count < 4; ++count)
7  {
8  deq.push_back(count); // вставляем числа в конец массива
9  deq.push_front(10 - count); // вставляем числа в начало массива
10 }
11 for (int index=0; index < deq.size(); ++index)
12 cout << deq[index] << ' ';
13 cout << '\n';
14 }
```

Результат выполнения программы выше:

**7 8 9 10 0 1 2 3**

**List** (или просто «**список**») — это двусвязный список, каждый элемент которого содержит 2 указателя: один указывает на следующий элемент списка, а второй — на предыдущий элемент списка. `list` предоставляет доступ только к началу и к концу списка — произвольный доступ запрещён. Если вы хотите найти значение где-то в середине, то вы должны начать с одного конца и перебирать каждый элемент списка до тех пор, пока не найдёте то, что ищите. Преимуществом двусвязного списка является то, что вставка элементов происходит очень быстро, если вы, конечно, знаете, куда хотите

вставлять. Обычно для перебора элементов двусвязного списка используются итераторы.

Вот функции, которые можно применять в своей программе вместе со списком:

**Таблица 11.1**

| | | |
|---|---|---|
| **pop_front** | удалить элемент в начале | |
| **pop_back** | удалить элемент в конце | |
| **push_front** | добавить элемент в начала | |
| **push_back** | добавить элемент в конец | |
| **front** | обратится к первому элементу | |
| **back** | обратиться к последнему элементу | |
| **insert(<позиция>, <значение>);** | добавить элемент в какое-то место | С помощью его можно добавить новый элемент в любую часть контейнера (в нашем случае для списка). Первым аргументом передаем — местоположение. Оно указывается итератором. Вторым значение новой ячейки. Здесь может быть как переменная так и просто значение. |

| | | |
|---|---|---|
| **copy(myspisok.begin(), myspisok.end(), ostream_iterator<int> (cout," "));** | вывести все элементы списка | Первые два значения (myspisok.begin(), myspisok.end()) которые должны передать, — это итераторы начала и конца контейнера. Дальше используем итератор вывода — ostream_iterator<int>(cout," "). В кавычках указывается значение между элементами (в нашем случае это пробел). |
| **unique** | удалить все дубликаты | Удаляет все повторяющиеся элементы (дубликаты). Использовать его очень просто: myspisok.unique(); |
| **merge** | добавление другого списка | Добавляет существующему списку еще один. myspisok.merge(dob_spisok); |

### §10.6.1.2 Ассоциативные контейнеры

Ассоциативные контейнеры — это контейнерные классы, которые автоматически сортируют все свои элементы (в том числе и те, которые вставляете вы). По умолчанию ассоциативные контейнеры выполняют сортировку элементов, используя оператор сравнения <.

- **set** — это контейнер, в котором хранятся только уникальные элементы, повторения запрещены. Элементы сортируются в соответствие с их значениями.
- **multiset** — это set, но в котором допускаются повторяющиеся элементы.

- **map** (или ещё «ассоциативный массив») — это set, в котором каждый элемент является парой «ключ-значение». Ключ используется для сортировки и индексации данных и должен быть уникальным. А значение — это фактические данные.
- **multimap** (или ещё «словарь») — это map, который допускает дублирование ключей. Все ключи отсортированы в порядке возрастания, и вы можете посмотреть значение по ключу.

Подробнее разберем работу с map. Для создания ассоциативного массива необходимо подключить соответствующую библиотеку <map>.

Чтобы создать map нужно воспользоваться данной конструкцией:

```
1  map < <L>, <R> > <имя>;
```

- <L> — этот тип данных будет относиться к значению ключа.
- <R> — этот тип данных соответственно относится к значению.

```
1  map <string, int> mp; // пример
```

В данном случае ключ – это строка, а значение – целочисленное значение.

При создании map все его элементы будут иметь значение нуля. Также имеется возможность добавить значения при инициализации:

```
1  map <string, string> book = {{"Hi", "Привет"},
2                               {"Student", "Студент"},
3                               {"!", "!"}};
```

Вывод значения производится путем обращения через перегруженный оператор индексации [] к ключу пары:

```
1  cout << book["Hi"];
```

Ниже мы разберем функции которые можно использовать для работы с map.

1. insert - это функция вставки нового элемента.

```
1  mp.insert(make_pair(num_1, num_2));
```

num_1 — ключ.

num_2 — значение.

`make_pair` – функция, использующаяся для создания пары элементов в ассоциативном массиве.

Мы можем сделать то же самое вот так:

```
1  mp[num_1] = num_2;
```

2. count – возвращает количество элементов с данным ключом. В нашем случае будет возвращать — 1 или 0. Эта функция больше подходит для multimap, у которого таких значений может быть много.

3. find – у этой функции основная цель узнать, есть ли определенный ключ в контейнере. Если он есть, то передать итератор на его местоположение. Если его нет, то передать итератор на конец контейнера. Например, давайте разберем данный код:

**Листинг 11.23**

```
1  #include <iostream>
2  #include <map>  // подключили библиотеку
3  using namespace std;
4  int main() {
5    setlocale(0, "");
6    map <string, string> book;
7    book["book"] = "книга";
8    map <string, string> :: iterator it, it_2;
9    it = book.find("book");
10   cout << it->second << endl;
11   it_2 = book.find("books");
12   if (it_2 == book.end()) {
13       cout << "Ключа со значением 'books' нет";
14   }
15   system("pause");
16   return 0;
17 }
```

4. erase – функция удаления пар из контейнера. Давайте посмотрим, как она работает на примере:

**Листинг 11.24**

```
1  map <string, string> passport
2  passport["maxim"] = "Denisov" // добавляем
3  passport["andrey"] = "Puzerevsky" // новые
4  passport["dima"] = "Tilyupo" // значения
5  cout << "Size: " << passport.size() << endl
6  map <string, string> :: iterator full_name // создали
   итератор на passport
```

```
7  full_name = passport.find("andrey") // находим ячейку
8  passport.erase(full_name)  // удаляем
9  cout << "Size: " << passport.size()
```

`passport` в нашем случае хранит имя как ключ, а фамилию как значение.

## §10.6.2 Итераторы STL

**Итератор** — это объект, который способен перебирать элементы контейнерного класса без необходимости пользователю знать реализацию определённого контейнерного класса. Во многих контейнерах (особенно в списке и в ассоциативных контейнерах) итераторы являются основным способом доступа к элементам этих контейнеров.

Об итераторе можно думать, как об указателе на определённый элемент контейнерного класса с дополнительным набором перегруженных операторов для выполнения чётко определённых функций:

- Оператор * возвращает элемент, на который в данный момент указывает итератор.
- Оператор ++ перемещает итератор к следующему элементу контейнера. Большинство итераторов также предоставляют оператор −− для перехода к предыдущему элементу.
- Операторы == и != используются для определения того, указывают ли два итератора на один и тот же элемент или нет. Для сравнения значений, на которые указывают два итератора, нужно сначала разыменовать эти итераторы, а затем использовать оператор == или !=.
- Оператор = присваивает итератору новую позицию (обычно начало или конец элементов контейнера). Чтобы присвоить значение элемента, на который указывает итератор, другому объекту, нужно сначала разыменовать итератор, а затем использовать оператор =.

Каждый контейнерный класс имеет 4 основных метода для работы с оператором =:

- begin() возвращает итератор, представляющий начало элементов контейнера.
- end() возвращает итератор, представляющий элемент, который находится после последнего элемента в контейнере.

- cbegin() возвращает константный (только для чтения) итератор, представляющий начало элементов контейнера.
- cend() возвращает константный (только для чтения) итератор, представляющий элемент, который находится после последнего элемента в контейнере.

Может показаться странным, что end() не указывает на последний элемент контейнера, но это сделано в целях упрощения использования циклов: цикл перебирает элементы до тех пор, пока итератор не достигнет end.

Наконец, все контейнеры предоставляют (как минимум) два типа итераторов:

- `container::iterator` — итератор для чтения/записи;
- `container::const_iterator` — итератор только для чтения.

Рассмотрим несколько примеров использования итераторов.

Заполним вектор 5-тью числами и, с помощью итераторов, выведем значения вектора:

**Листинг 11.25**

```
1  #include <iostream>
2  #include <vector>
3  int main()
4  {
5  vector<int> myVector;
6  for (int count=0; count < 5; ++count)
7  myVector.push_back(count);
8  vector<int>::const_iterator it; // объявляем итератор только для чтения
9  it = myVector.begin(); // присваиваем ему начало вектора
10 while (it != myVector.end()) // пока итератор не достигнет конца
11 {
12 cout << *it << " "; // выводим значение элемента, на который указывает итератор
13 ++it; // и переходим к следующему элементу
14 }
15 cout << '\n';
16 }
```

Результат выполнения программы выше:

**0 1 2 3 4**

Контейнеры `map` и `multimap` принимают пары элементов (определённых как `std::pair`). Мы используем вспомогательную функцию **make_pair()** для создания пар. `std::pair` позволяет получить доступ к элементу (паре ключ-значение) через первый и второй члены. В нашем ассоциативном массиве мы используем первый член в качестве ключа, а второй — в качестве значения:

**Листинг 11.26**

```
1  #include <iostream>
2  #include <map>
3  #include <string>
4  int main()
5  {
6  map<int, std::string> myMap;
7  myMap.insert(std::make_pair(3, "cat"));
8  myMap.insert(std::make_pair(2, "dog"));
9  myMap.insert(std::make_pair(5, "chicken"));
10 myMap.insert(std::make_pair(4, "lion"));
11 myMap.insert(std::make_pair(1, "spider"));
12 map<int, std::string>::const_iterator it; // объявляем итератор
13 it = myMap.begin(); // присваиваем ему начало вектора
14 while (it != myMap.end()) // пока итератор не достигнет конца
15 {
16 cout << it->first << "=" << it->second << " "; // выводим значение элемента, на который указывает итератор
17 ++it; // и переходим к следующему элементу
18 }
19 cout << '\n';
20 }
```

Результат выполнения программы выше:

**1=spider 2=dog 3=cat 4=lion 5=chicken**

Итераторы предоставляют простой способ перебора элементов контейнерного класса без необходимости знать реализацию определённого

контейнерного класса. В сочетании с алгоритмами STL и методами контейнерных классов итераторы становятся ещё более мощными.

### §10.6.3 Алгоритмы STL

Алгоритмы STL реализованы в виде **глобальных** функций, которые работают с использованием итераторов. Это означает, что каждый алгоритм нужно реализовать всего лишь один раз, и он будет работать со всеми контейнерами, которые предоставляют набор итераторов (включая и ваши собственные (пользовательские) контейнерные классы). Хотя это имеет огромный потенциал и предоставляет возможность быстро писать сложный код, у алгоритмов также есть и «тёмная сторона»: некоторая комбинация алгоритмов и типов контейнеров может не работать, работать с плохой производительностью или вызывать бесконечные циклы. Поэтому следует быть осторожным.

STL предоставляет довольно много алгоритмов, в рамках данной дисциплины рассмотрим только наиболее распространённые. Для их работы нужно подключить **заголовочный файл** algorithm.

Алгоритмы `min_element()` и `max_element()` находят минимальный и максимальный элементы в контейнере и возвращают итератор (указатель) на найденные элементы:

**Листинг 11.27**

```
 1  #include <iostream>
 2  #include <list>
 3  #include <algorithm>
 4  int main()
 5  {
 6  list<int> li;
 7  for (int nCount=0; nCount < 5; ++nCount)
 8  li.push_back(nCount);
 9  list<int>::const_iterator it; // объявляем итератор
10  it = min_element(li.begin(), li.end());
11  cout << *it << ' ';
12  it = max_element(li.begin(), li.end());
13  cout << *it << ' ';
14  cout << '\n';
15  }
```

Результат выполнения программы выше:

**0 4**

В следующем примере мы используем алгоритм find(), чтобы найти определённое значение в списке, а затем используем функцию list::insert() для добавления нового значения в список:

**Листинг 11.28**

```
1  #include <iostream>
2  #include <list>
3  #include <algorithm>
4  int main()
5  {
6  list<int> li;
7  for (int nCount=0; nCount < 5; ++nCount)
8  li.push_back(nCount);
9  list<int>::iterator it; // объявляем итератор
10 it = find(li.begin(), li.end(), 2); // ищем в списке число 2
11 li.insert(it, 7); // используем list::insert для вставки числа 7 перед числом 2
12 for (it = li.begin(); it != li.end(); ++it) // выводим с помощью цикла и итератора элементы списка
13 cout << *it << ' ';
14 cout << '\n';
15 }
```

Результат выполнения программы выше:

**0 1 7 2 3 4**

В следующем примере мы отсортируем весь вектор, а затем выполним реверс его элементов:

**Листинг 11.29**

```
1  #include <iostream>
2  #include <vector>
3  #include <algorithm>
4  int main()
5  {
6  vector<int> vect;
7  vect.push_back(4);
8  vect.push_back(8);
9  vect.push_back(-3);
10 vect.push_back(3);
```

```
11  vect.push_back(-8);
12  vect.push_back(12);
13  vect.push_back(5);
14  sort(vect.begin(), vect.end()); // выполняем сортировку элементов вектора
15  vector<int>::const_iterator it; // объявляем итератор
16  for (it = vect.begin(); it != vect.end(); ++it) // выводим с помощью цикла и итератора элементы вектора
17  cout << *it << ' ';
18  cout << '\n';
19  reverse(vect.begin(), vect.end()); // выполняем реверс элементов вектора
20  for (it = vect.begin(); it != vect.end(); ++it) // выводим с помощью цикла и итератора элементы вектора
21  cout << *it << ' ';
22  cout << '\n';
23  }
```

Результат выполнения программы выше:

**-8 -3 3 4 5 8 12**

**12 8 5 4 3 -3 -8**

Рассмотрим пример программы, где используется все компоненты стандартной библиотеки шаблонов.

Постановка задачи: создать контейнер, содержащий основную информацию о студентах университета (Фамилия, возраст, номер группы). Свойства объекта «Студент» должны быть инкапсулированы в классе. Реализовать добавление новых студентов, удаление и сортировку по заданным свойствам.

Для начала реализуем класс «Студент». В данном классе будет три закрытых свойства: name (фамилия студента), age (возраст), numgroup (номер группы), конструктор student() для создания экземпляра данного класса с заданными пользователем значениями и методы доступа для чтения закрытых свойств класса

**Листинг 11.30**

```
1  class student
2  {
```

```
3     private:
4         string name;
5         int age;
6         int numgroup;
7     public:
8         student(string name, int age, int numgroup)
9         {
10            this->age=age;
11            this->numgroup=numgroup;
12            this->name=name;
13        }
14        string getname()
15        {
16            return name;
17        }
18        int getage()
19        {
20            return age;
21        }
22        int getnumgroup()
23        {
24            return numgroup;
25        }
26 };
```

Так же объявим глобальную переменную flag, которая будет выполнять роль переключателя в дальнейшем описании кода программы.

```
1 int *flag = new int
```

В качестве контейнера для хранения объектов-студентов будем использовать вектор. На место типа данных ставим имя класса (пользовательский тип данных). Объявление вектора будет выглядеть следующим образом:

```
1 vector <student> vecstud;
```

Напишем отдельную функцию для добавления студентов.

**Листинг 11.31**

```
1 void addstudent(vector <student> *vecstud, string
  name, int age, int numgroup)
2 {
3     (*vecstud).push_back(student(name, age,
  numgroup));
```

```
4  }
```

В качестве параметров функции передаем адрес на вектор-контейнер, имя студента, добавляемого в контейнер, значение свойства «Возраст» и «Номер группы». Добавление происходит в конец вектора, с помощью метода `push_back`. Место параметра данного метода занимает конструктор класса, так как вектор хранит объекты типа `student` (конструктор возвращает объект типа `student`).

Функция для отображения всех объектов, содержащихся в контейнере, примет вид:

**Листинг 11.32**

```
1  void showstudent(vector <student> *vecstud)
2  {
3      for(int i = 0; i < (*vecstud).size(); i++)
4      {
5          cout << (*vecstud)[i].getname() << "\t" <<
   (*vecstud)[i].getage() << "\t" <<
   (*vecstud)[i].getnumgroup() << "\t" << endl;
6      }
```

Здесь функция `showstudent` принимает как параметр лишь адрес вектора, обработка которого будет происходить в данном участке кода. Обработка всех ячеек вектора происходит в цикле `for`, обращение к элементам вектора происходит через перегруженный оператор индексации [].

Для удаления элемента по заданной фамилии сначала объявим итератор для вектора, чтобы получить доступ к любому месту контейнера.

```
1  vector <student>::iterator it
```

Далее совершим запрос у пользователя с целью поиска элемента в векторе по заданному значению фамилии студента. Вызываем функцию удаления элемента, передав как аргумент функции итератор, адрес вектора и искомое значение свойства «Фамилия». Реализация функции удалении представлена ниже:

**Листинг 11.33**

```
1  void deletestudent(vector <student>::iterator it,
   vector <student> *vecstud, string name)
```

```
2  {
3      for(it = (*vecstud).begin(); it != (*vecstud).end(); it++)
4      {
5          if((*it).getname() == name)
6          {
7              cout << "Объект " << name << " отчислен" << endl;
8              (*vecstud).erase(it);
9              break;
10         }
11     }
12 }
```

Объявляем цикл for, границы которого заданы итераторами на первый и следующий после последнего элемент. Ищем совпадение значения, которое возвращает «разыменованный» итератор, и искомого элемента. Если поиск успешен, то выводим служебное сообщение и удаляем ячейку из вектора с помощью метода `erase`.

Теперь самое непростое – сортировка элементов по заданному значению. Когда происходит сортировка контейнера объектов пользовательского типа с использованием метода sort напрямую:

```
1  sort(vecstud.begin(), vecstud.end());
```

Произойдет ошибка, так как в данном методе не задано правило сортировки для объектов типа student. Необходимо перегрузить оператор сравнения < со своим правилом сравнения, чтобы метод смог выполнять сортировку нашего вектора. Чтобы выполнять сортировки определенного свойства, используем ветвление с использованием ранее созданной переменной-переключателя.

**Листинг 11.34**

```
1  cout << "Выберите, по какому свойству отсортировать студентов: 1) Фамилия; 2)Возраст; 3) Номер группы" << endl;
2  cin >> *flag;
3  sort(vecstud.begin(), vecstud.end());
```

Перегруженный оператор сравнения имеет следующий вид:

**Листинг 11.35**

```
1  bool operator < (student &lo, student &ro)
2  {
3      if((*flag) == 1)
4      {
5          return (lo.getname() < ro.getname());
6      }
7      else if((*flag) == 2)
8      {
9          return (lo.getage() < ro.getage());
10     }
11     else if((*flag) == 3)
12     {
13         return (lo.getnumgroup() < ro.getnumgroup());
14     }
15 }
```

Перегруженный оператор представляет собой функцию, принимающую в качестве параметров левый операнд и правый операнд операции сравнения. Оба операнда имеют тип данных `student` – наши объекты-студенты.

Осталось организовать удобный пользовательский интерфейс для работы с программой:

**Листинг 11.36**

```
1  int main()
2  {
3  SetConsoleCP(1251);
4  SetConsoleOutputCP(1251);
5  string name;
6  int age, numgroup;
7  vector <student> vecstud;
8  while(true)
9  {
10 system("cls");
11 cout << "1 - Добавить студента в контейнер" << endl;
12 cout << "2 - Вывести всех студентов" << endl;
13 cout << "3 - Отчислить студента" << endl;
14 cout << "4 - Отсортировать студентов" << endl;
15 cout << "5 - Выйти из программы" << endl;
16 cin >> *flag;
```

```
17 switch(*flag)
18 {
19 case 1:
20 {
21 cout << "Фамилия студента: ";
22 cin >> name;
23 cout << endl << "Возраст студента: ";
24 cin >> age;
25 cout << endl << "Номер группы: ";
26 cin >> numgroup;
27 addstudent(&vecstud, name, age, numgroup);
28 break;
29 }
30 case 2:
31 {
32 showstudent(&vecstud);
33 Sleep(5000);
34 break;
35 }
36 case 3:
37 {
38 vector <student>::iterator it;
39 cout << "Введите ФИО студента, которого хотите отчислить: " << endl;
40 cin >> name;
41 deletestudent(it, &vecstud, name);
42 break;
43 }
44 case 4:
45 {
46 cout << "Выберите, по какому свойству отсортировать студентов: 1) Фамилия; 2)Возраст; 3) Номер группы" << endl;
47 cin >> *flag;
48 sort(vecstud.begin(), vecstud.end());
49 break;
50 }
51 case 5:
52 {   exit(0);    }
53 default:
54 {
```

```
55  cout << "Некорректный ввод, повторите попытку." << endl;
56  break;
57  }}}}
```

Для работы с данной программой необходимо подключить следующие заголовочные файлы:

**Листинг 11.37**

```
1  #include <iostream>
2  #include <ctime>
3  #include <vector>
4  #include <string>
5  #include <windows.h>
6  #include <algorithm>
```

## §10.7 ArrayList

Хотя в языке C# есть массивы, которые хранят в себе наборы однотипных объектов, но работать с ними не всегда удобно. Например, массив хранит фиксированное количество объектов, однако, что, если мы заранее не знаем, сколько нам потребуется объектов. И в этом случае намного удобнее применять коллекции. Еще один плюс коллекций состоит в том, что некоторые из них реализует стандартные структуры данных, например, стек, очередь, словарь, которые могут пригодиться для решения различных специальных задач.

Большая часть классов коллекций содержится в пространствах имен:

1. System.Collections (простые необобщенные классы коллекций);
2. System.Collections.Generic (обобщенные или типизированные классы коллекций);
3. System.Collections.Specialized (специальные классы коллекций).

Итак, класс ArrayList представляет коллекцию объектов. И если надо сохранить вместе разнотипные объекты - строки, числа и т.д., то данный класс как раз для этого подходит.

Основные методы класса:

**Таблица 11.2**

| **Метод** | **Описание метода** |
|---|---|
| int Add(object value) | добавляет в список объект value |
| void AddRange(ICollection col) | добавляет в список объекты коллекции col, которая представляет интерфейс ICollection - интерфейс, реализуемый коллекциями. |
| void Clear() | удаляет из списка все элементы |
| bool Contains(object value) | проверяет, содержится ли в списке объект value. Если содержится, возвращает true, иначе возвращает false |
| void CopyTo(Array array) | копирует текущий список в массив array |
| ArrayList GetRange(int index, int count) | возвращает новый список ArrayList, который содержит count элементов текущего списка, начиная с индекса index |
| int IndexOf(object value) | возвращает индекс элемента value |
| void Insert(int index, object value) | вставляет в список по индексу index объект value |
| void InsertRange(int index, ICollection col) | вставляет в список начиная с индекса index коллекцию ICollection |
| int LastIndexOf(object value) | возвращает индекс последнего вхождения в списке объекта value |
| void Remove(object value) | удаляет из списка объект value |
| void RemoveAt(int index) | удаляет из списка элемент по индексу index |
| void RemoveRange(int index, int count) | удаляет из списка count элементов, начиная с индекса index |

| | |
|---|---|
| void Reverse() | переворачивает список |
| void SetRange(int index, ICollection col) | копирует в список элементы коллекции col, начиная с индекса index |
| void Sort() | сортирует коллекцию |

Кроме того, с помощью свойства Count можно получить количество элементов в списке.

Посмотрим применение класса на примере.

**Листинг 11.38**

```
1  using System;
2  using System.Collections;
3  namespace Collections
4  {
5      class Program
6      {
7          static void Main(string[] args)
8          {
9              ArrayList list = new ArrayList();
10             list.Add(2.3); // заносим в список объект типа double
11             list.Add(55); // заносим в список объект типа int
12             list.AddRange(new string[] { "Hello", "world" }); // заносим в список строковый массив
13             // перебор значений
14             foreach (object o in list)
15             {
16                 Console.WriteLine(o);
17             }
18             // удаляем первый элемент
19             list.RemoveAt(0);
20             // переворачиваем список
21             list.Reverse();
22             // получение элемента по индексу
23             Console.WriteLine(list[0]);
24             // перебор значений
25             for (int i = 0; i < list.Count; i++)
26             {
27                 Console.WriteLine(list[i]);
28             }
```

```
29              Console.ReadLine();
30          }
31      }
32  }
```

Во-первых, так как класс `ArrayList` находится в пространстве имен `System.Collections`, то подключаем его (`using System.Collections;`).

Вначале создаем объект коллекции через метод-конструктор как объект любого другого класса:

```
ArrayList list = new ArrayList();
```

При необходимости мы могли бы так же, как и с массивами, выполнить начальную инициализацию коллекции, например,

```
ArrayList list = new ArrayList(){1, 2, 5, "string", 7.7};
```

Далее последовательно добавляем разные значения. Данный класс коллекции, как и большинство других коллекций, имеет два способа добавления: одиночного объекта через метод `Add` и набора объектов, например, массива или другой коллекции через метод `AddRange`.

Через цикл `foreach` мы можем пройтись по всем объектам списка. И поскольку данная коллекция хранит разнородные объекты, а не только числа или строки, то в качестве типа перебираемых объектов выбран тип `object`:

```
foreach (object o in list)
```

Многие коллекции, в том числе и `ArrayList`, реализуют удаление с помощью методов `Remove/RemoveAt`. В данном случае мы удаляем первый элемент, передавая в метод `RemoveAt` индекс удаляемого элемента.

В завершении мы опять же выводим элементы коллекции на экран только уже через цикл `for`. В данном случае с перебором коллекций дело обстоит также, как и с массивами. А число элементов коллекции мы можем получить через свойство `Count`.

С помощью индексатора мы можем получить по индексу элемент коллекции так же, как и в массивах:

```
object firstObj = list[0]
```

## §10.8 Список List

Класс List<T> из пространства имен System.Collections.Generic представляет простейший список однотипных объектов.

Среди его методов можно выделить следующие:

**Таблица 11.3**

| Метод | Описание метода |
|---|---|
| void Add(T item) | добавление нового элемента в список |
| void AddRange(ICollection collection) | добавление в список коллекции или массива |
| int BinarySearch(T item) | бинарный поиск элемента в списке. Если элемент найден, то метод возвращает индекс этого элемента в коллекции. При этом список должен быть отсортирован |
| int IndexOf(T item) | возвращает индекс первого вхождения элемента в списке |
| void Insert(int index, T item) | вставляет элемент item в списке на позицию index |
| bool Remove(T item) | удаляет элемент item из списка, и если удаление прошло успешно, то возвращает true |
| void RemoveAt(int index) | удаление элемента по указанному индексу index |
| void Sort() | сортировка списка |

Посмотрим реализацию списка на примере:

**Листинг 11.39**

```
1  using System;
2  using System.Collections.Generic;
3  namespace Collections
4  {
5      class Program
6      {
```

```
7         static void Main(string[] args)
8         {
9             List<int> numbers = new List<int>() { 1, 2, 3, 45 };
10            numbers.Add(6); // добавление элемента
11            numbers.AddRange(new int[] { 7, 8, 9 });
12            numbers.Insert(0, 666); // вставляем на первое место в списке число 666
13            numbers.RemoveAt(1); //  удаляем второй элемент
14            foreach (int i in numbers)
15            {
16                Console.WriteLine(i);
17            }
18            List<Person> people = new List<Person>(3);
19            people.Add(new Person() { Name = "Том" });
20            people.Add(new Person() { Name = "Билл" });
21            foreach (Person p in people)
22            {
23                Console.WriteLine(p.Name);
24            }
25            Console.ReadLine();
26        }
27    }
28    class Person
29    {
30        public string Name { get; set; }
31    }
32 }
```

Здесь у нас создаются два списка: один для объектов типа int, а другой - для объектов `Person`. В первом случае мы выполняем начальную инициализацию списка:

```
List<int> numbers = new List<int>() { 1, 2, 3, 45 };
```

Во втором случае мы используем другой конструктор, в который передаем начальную емкость списка:

```
List<Person> people = new List<Person>(3);
```

Указание начальной емкости списка (capacity) позволяет в будущем увеличить производительность и уменьшить издержки на выделение памяти при добавлении элементов. Также начальную емкость можно установить с помощью свойства `Capacity`, которое имеется у класса `List`.