ТЕХНОЛОГИИ ПРОГРАММИРОВАНИЯ
(6 СЕМЕСТР)
ЛАБОРАТОНАЯ РАБОТА №1.3
ФУНКЦИИ

Скирневский И.П.

Томск – 2016

ОГЛАВЛЕНИЕ

ЛАБАРАТОРНАЯ №1.3. ФУНКЦИОНАЛЬНЫЙ ПОДХОД ~~– Это без небольшого кусока начало раздела 3 про функциональный подход ☺~~

1.1.Ключевые особенности функционального подхода

Первыми формами модульности, появившимися в языках программирования, были процедуры и функции. Поскольку функции в математике использовались издавна, то появление их в языках программирования было совершенно естественным. Уже с первых шагов программирования процедуры и функции позволяли решать одну из важнейших задач, стоящих перед программистами, – задачу повторного использования программного кода. Один раз написанную функцию можно многократно вызывать в программе с разными значениями параметров, передаваемых функции в момент вызова. Встроенные в язык функции позволяли существенно расширить возможности языка программирования. Важным шагом в автоматизации программирования было появление библиотек процедур и функций, доступных из языка программирования.

С появлением ООП архитектурная роль функциональных модулей отошла на второй план. Для ОО-языков, к которым относится и язык C#, роль архитектурного модуля играет класс. Программная система строится из модулей, роль которых играют классы, но каждый из этих модулей имеют содержательную начинку, задавая некоторую абстракцию данных.

Процедуры и функции связываются теперь с классом, они обеспечивают требуемую функциональность класса и называются методами класса. Поскольку класс в объектно-ориентированном программировании рассматривается как некоторый тип данных, то главную роль в классе начинают играть его данные – поля класса, задающие свойства объектов класса. Методы класса «служат» данным, занимаясь их обработкой. Помните, в C# процедуры и функции существуют только как методы некоторого класса, они не существуют вне класса.

В данном контексте понятие класс распространяется и на все его частные случаи – структуры, интерфейсы, делегаты.

В языке C# нет специальных ключевых слов – method, procedure, function, но сами понятия конечно же присутствуют. Синтаксис объявления метода позволяет однозначно определить, чем является метод – процедурой или функцией.

1.2. Инициализация функции Main

Как вы могли заметить, в предыдущей лабораторной работе, весь код нашего приложения был написан в рамках одной единственной функции – Main. Функциональных подход предполагает разложение кода над ряд примитивов выполняющих определенное действие – функций. Таким образом задачей третьей лабораторной работы будет вынесение простых пользовательских операций в функции.

Как вы считаете, какие основные плюсы функционального подхода? Во-первых это примитивная декомпозиция кода на более мелкие куски, что позволяет упростить читабельность. Но, ключевое преимущество вынесения отдельных блоков в функции – это повторное использование кода. Так, используя структурный подход нам бы пришлось каждый раз писать код добавляющий строку в массив или выполнять эту операция в цикле. Давайте исправим эту избыточность в третьей лабораторной работе.

Примечание. С остальными плюсами функционального подхода вы познакомитесь прочитав лекционный материал курса.

Как уже было упомянуто ранее, тело всей программы во второй лабораторной работе находилось внутри одной функции Main. Вспомним, как выглядела наша функция изучив код ниже.

```
static void Main(string[] args)
{
   Какой-то код
}
```

И так, приведенная выше строка инициализирует функцию Main. Как показано на рисунке 21 **тело или сигнатура** функции состоит из следующих ключевых элементов: возвращаемый тип, имя функции, параметры или аргументы функции. Поле модификатор не относится непосредственно к функции, поэтому мы не будем его рассматривать, а лишь запомним, что в первых лабораторных работах, все функции будут иметь модификатор static.

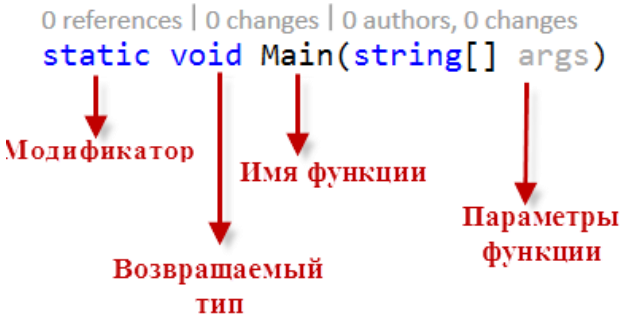

Рисунок 1 – Сигнатура (тело) функции

Важно!

Функция в языке C#, (как в прочем и любом другом языке программирования) отражает какой-либо процесс. Например, процесс сложения двух чисел можно назвать функцией Add, процесс вывода информации на экран Print и так далее. Стоит отметить, что неправильно будет называть функцию Length (Длина), так как слово length не отражает никакого процесса или действие. Верное название для подобной функции может быть, например, GetLength (Получить длину) или SetLength (Установить длину).

1.3. Возвращаемый тип

Каждая функция либо:

– Возвращает что-то (**int** Sum (), **int** GetLenght (), **string** GetName())

– Ничего не возвращает (**void** Main (), **void** Print (), **void** Draw (), **void** Format())

Каждая функция может либо возвращать внешнему миру (вызывающему коду) какие-то данные, либо просто выполнять какую-либо задачу. Предположим, у нас есть функция Sum, которая выполнят сложение двух чисел, очевидно, что такая функция должна вернуть число, равное этой сумме, тоже касается и функции GetLength (получить длину), которая возвращает число равное длине (например, длине диагонали вашего телевизора).

С другой стороны функция Print() просто выводит какие-то данные на консоль и ей не требуется ничего возвращать, функция Drow() может просто рисовать квадрат на холсте и ей тоже нечего вернуть внешнему миру. Наверно многим уже стало понятно, что при работе с функциями, которая ничего не возвращают используется ключевое слово **void** (пустота), а для функций с возвращаемым значением указывается необходимый тип данных (int, string и др.).

1.4. Имя функции

Мы не будем подробно останавливаться на данной части сигнатуры функции – очевидно, что имя функции используется для дальнейшего обращения к созданной функции по коду проекта. Стоит отметить, что все функции должны начинаться с большой буквы. (пример: Main, Sum, Print, Drow). Если имя функции состоит из двух слов, используется стиль написания CamelCase при котором несколько слов пишутся слитно без пробелов, при этом каждое слово пишется с заглавной буквы.

Примечание. Стиль получил название CamelCase, поскольку заглавные буквы внутри слова напоминают горбы верблюда. (пример: DrowImage, PrintName).

1.5. Параметры функции

Если вы уже читали лекционный материал курса, то можете помнить, что список формальных аргументов метода может быть пустым. Хотя в случае с функцией Main это не так. Пустой список параметров или аргументов – довольно типичная ситуация. Список может содержать фиксированное число аргументов, разделяемых символом запятой.

Рассмотрим синтаксис объявления одного формального аргумента:

[ref|out|params] тип_аргумента имя_аргумента

Обязательным является указание типа и имени аргумента. Заметьте, никаких ограничений на тип аргумента не накладывается. Он может быть любым скалярным типом, массивом, классом, структурой, интерфейсом, перечислением, функциональным типом.

Несмотря на фиксированное число формальных аргументов, есть возможность при вызове метода передавать ему **произвольное число фактических аргументов**. Для реализации этой возможности в списке формальных аргументов необходимо задать ключевое слово *params*. Оно может появляться в объявлении лишь для последнего аргумента списка, объявляемого как массив произвольного типа. При вызове метода этому формальному аргументу соответствует произвольное число фактических аргументов.

Содержательно, все аргументы метода разделяются на три группы: **входные, выходные и обновляемые**. Аргументы первой группы передают информацию методу, их значения в теле метода только читаются. Аргументы второй группы представляют собой результаты метода, они получают значения в ходе работы метода. Аргументы третьей группы выполняют обе функции. Их значения используются в ходе вычислений и обновляются в результате работы метода. Выходные аргументы всегда должны сопровождаться ключевым словом out, обновляемые – ref. Что же касается входных аргументов, то, как правило, они задаются без ключевого слова, хотя иногда их полезно объявлять с параметром ref, о чем мы будет говорить позже. Заметьте, если аргумент объявлен как выходной с ключевым словом out, то в теле метода обязательно должен присутствовать оператор присваивания, задающий значение этому аргументу. В противном случае возникает ошибка еще на этапе компиляции.

1.6. Тело функции

Синтаксически тело метода является блоком, представляющим последовательность операторов и описаний переменных, заключенную в фигурные скобки. Если речь идет о теле функции, то в блоке должен быть хотя бы один оператор, возвращающий значение функции в форме *return <выражение>*.

Переменные, описанные в блоке, считаются локализованными в этом блоке. В записи операторов блока участвуют имена локальных переменных блока, имена полей класса и имена аргументов метода.

1.7. Структуры

В данной лабораторной работе, по мимо знакомства с функциями, студентом предлагается реализовать хранилище данных в виде массива структур. Ниже дан краткий обзор понятия структуры в языке C#.

В .NET. Типы структур хорошо подходят для моделирования математических, геометрических и других “атомарных” сущностей в приложении (в нашем случая структура будет моделировать объект – Запись). Структура – это определяемый пользователем тип; структуры представляют собой типы, которые могут содержать любое количество полей данных и членов, оперирующих над этими полями.

Примечание. Если вы имеете опыт объектно-ориентированного программирования, можете считать структуры "облегченными классами", т.к. они предоставляют способ определения типа, поддерживающего инкапсуляцию, но не могут применяться для построения семейства взаимосвязанных типов. Когда возникает потребность в создании семейства типов, связанных через наследование, необходимо использовать классы.

На первый взгляд процесс определения и использования структур выглядит очень просто, но, как известно, сложности обычно скрываются в деталях. В C# структуры определяются с применением ключевого слова **struct**. Определим новую структуру под названием **Note**, содержащую две переменных-члена типа string и DataTime и набор методов для взаимодействия с ними.

```
class Program
{
    struct Note
    {
        public string text;
        public DateTime date;

        public Note(string note)
        {
            text = note;
            date = DateTime.Now;
        }
    }
```

До этого, как вы помните наше приложение позволяем хранить запись в виде строки символов. Предположим, что мы хотим иметь возможность хранить помимо самой записи время ее добавления. Для этого мы могли бы создать дополнительные переменные-списки, однако, это значительно усложнит нашу программу, поскольку мы будем вынуждены контролировать, чтобы номер записи в списке соответствовал номеру даты в списке и т.д. Но с помощью структуры мы можем в одном списка, в каждой записи хранить несколько значений. Теперь каждый элемент нашего списка будет хранить структуру Note, которая в свою очередь хранит информацию о дате и само сообщение. Надеюсь вы не запутались во вложенности.

Примечание. Код структуры должен располагаться до функции Main..

1.7.1. Конструктор структуры

Для того чтобы каждый раз во время добавления не задавать вручную дату, создадим конструктор, который будет автоматически записывать дату в поле date. Заметьте, что поля и конструктор объявлены с модификатором public, для того чтобы к ним можно было обратиться извне.

Конструктор структуры — это специальная функция, которая не имеет типа возвращаемого значения и не помечается ключевым словом void, фактически эта функция вызывается единожды, при создании структуры и служит некой отправной точкой. В рамках конструктора обычно производят различные настройки, в нашем случае мы выполняем инициализацию двух переменных.

```
        public Note(string note)
        {
            text = note;
            date = DateTime.Now;
        }
```

1.8. Модернизация приложения «Записная книжка»

После небольшого теоретического обзора модернизируем нашу лабораторную работу. Во-первых, необходимо отредактировать объявление списка записей.

```
List<Note> notes = new List<Note>();
```

Обращаю ваше внимание, что теперь список содержит объекты Note. В связи с этим изменим и код добавления записей.

```
notes.Add(new Note("Первая запись"));
notes.Add(new Note("Вторая запись"));
```

На данный момент наше приложение не использует возможности функций, не считая функцию Main. Поскольку функции могут использоваться не только для сокращения повторяющихся участков кода, но и для повышения читаемости кода, мы вынесем наши операции в блоке switch в отдельные функции.

Как мы уже говорили, в C# наименования функций принято оформлять в Pascal case стиле, когда каждое слово начинается с заглавной буквы, например, «GetNewMessage».

Вынесем в отдельную функцию код, отвечающий за вывод списка записей.

```
static void NotesList(List<Note> notes)
{
    foreach (var note in notes)
    {
        Console.WriteLine((notes.IndexOf(note) + 1) + ") " + note.text + " (" +
note.date.ToString("dd MMM HH:mm:ss") + ")"); // т.к. в списке нумерация начинается
с 0, то прибавляем 1
    }
}
```

Заметьте, что при выводе даты необходимо задать формат. В нашем случае это день, месяц, часы, минуты и секунды.

В качестве аргумента мы передаем наш список записей. Стоить заметить, что хоть аргумент и имеет имя notes он вовсе не привязан к переменной notes в главной функции Main. Другими словами, мы можем заменить notes на любое другой доступное имя переменной, правда, тогда придется отредактировать тело функции.

Заметьте, что для вывода индекса записи использована конструкция «notes.IndexOf(note) + 1». Студентам можно создать дополнительное поле в структуре с номером записи.

Вынесем функционал нашей программы в отдельные функции. А именно:

Функция отображения справки.

```
static void ShowHelp()
{
    Console.WriteLine("Доступные команды:");
    Console.WriteLine("a - добавить запись");
    Console.WriteLine("d - удалить запись с номером n");
    Console.WriteLine("h - список доступных команд");
    Console.WriteLine("l - список всех записей");
}
```

Функция добавления записи.

```
static void AddNote(List<Note> notes)
{
    Console.Write("Введите сообщение: ");
    var newNote = Console.ReadLine(); // считываем сообщение
    notes.Add(new Note(newNote)); // добавляем сообщение в конец списка
}
```

Функция удаления записи.

```
static void DeleteNote(List<Note> notes)
{
    Console.Write("Введите номер записи для удаления: ");
    int n = Convert.ToInt32(Console.ReadLine()) - 1;
    // проверим существует ли введеный номер записи
    if (n < notes.Count && n > -1)
    {
        notes.RemoveAt(n);
    }
    else
    {
        Console.WriteLine("Записи с указанным номером не существует");
    }
}
```

Самостоятельно изучите структуру функций, обращая внимание на синтаксис обращения к полям структуры Note.

Финальной частью является изменение блока switch. Теперь в каждом узле будут вызываться соответствующие функции. Финальный вариант блока switch приведен ниже.

```
switch (action)
{
    // вывести все записи
    case 'l':
        NotesList(notes);
        break;

    case 'h':
        ShowHelp();
        break;

    // добавить запись
    case 'a':
        AddNote(notes);
        break;

    // удалить запись
    case 'd':
        DeleteNote(notes);
        break;

    default:
        Console.WriteLine("Неизвестная команда");
        break;
}
```

Как мы видим блок switch стал более лаконичным. Читаемость кода значительно возрастает и ко всему прочему, созданные выше функции, можно будет использовать в дальнейшем, что значительно сократит дублирование кода.

На этом переход на функциональный или процедурный подход завершена. Код программы состоит из набора функций и операторов вызова данных функций.

Лабораторная номер 3 завершена.

Задание на лабораторную работу №3

1. Модернизировать консольное приложение, созданное в лабораторной работе №2, путем выноса основных операций в функции.

2. Функции должны иметь осмысленные названия, т.е. отображающие суть функции.

3. Создать структуру, содержащую необходимое количество полей.

4. Заменить изначальный список на список структур.

5. Оформить отчет с описанием хода выполнения вашей лабораторной работы.

6. Для четных вариантов (2,4,6 и т.д.) реализовать функцию поиска, которая ищет запись по заданному полю, результат поиска выводит на экран

7. Для не четных вариантов (1,3,5 и т.д.) реализовать функцию поиска, которая выполняет сквозной поиск по всем полям и выводит результат поиска на экран

8. В соответствии с вариантом выполнить дополнительное задание (**пример**: если у вас 7 вариант, вы выполняете 1 задание. Если у вас 14 вариант, вы выполняете 2 задание):

Вар 1. Реализовать функцию, которая выполняет сортировку записей по одному из полей и выводу их на экран

Вар 2. Реализовать функцию, которая выводит запрашиваемое количество записей, отсортированных по одному из полей

Вар 3. Реализовать функцию, которая позволяет добавить диапазон записей

Вар 4. Реализовать функцию, которая позволяет удалить диапазон записей

Вар 5. Реализовать функцию, которая позволяет обновить запись

Вар 6. Реализовать функцию, которая меняет регистр шрифта для выбранного пользователем поля записи

Требования к оформлению отчета

1 Формат листа – А4 (210 х 297 мм). Поля: слева – 25 мм; справа – 15 мм; сверху, снизу – 20 мм.

2 Основной текст документа должен быть напечатан шрифтом Times New Roman Cyr, размером 12 пт. Межстрочный интервал – 1.5, отступ красной строки – 1.25. Не добавлять разрыв между абзацами одного стиля. Выравнивание по ширине.

3 Размер заголовков – 12 пт. Заголовок первого уровня – Все заглавные, жирный, заголовок второго и последующих уровне – жирные, как в предложении. Отступ заголовка – 1.25. Выравнивание по левому краю.

4 Нумерация страниц производится автоматически.

5 Если позволяет структура документа, автоматически оформить оглавление.

6 Титульный лист оформлен по стандарту ТПУ.

7 Рекомендуемая структура отчета: Титульный лист, Оглавление (если требуется) Введение, Ход работы, Заключение. В разделе ход работы описывается последовательность выполнения заданий, если надо приводятся участки кода и скриншоты.

Требования к защите

Лабораторная считается защищенной при наличии трех файлов:

1. Архив с исходным кодом
2. Исполняемый файл
3. Отчет по лабораторной работе