ФГАОУ ВО «КФУ им. В.И. Вернадского»
Физико-технический институт (структурное подразделение)

Кафедра компьютерной инженерии и моделирования

Скибинский Дмитрий Константинович

отчет по практической работе №4
по дисциплине **«ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ»**

Направление подготовки:
09.03.04 "Программная инженерия"

Оценка 100

Симферополь, 2024

Практическая работа №4
Тема: Типы перечислений и структуры

Цель работы: Научиться на практике создавать перечисления и структуры. Разобраться самостоятельно с эффективностью использования структур, недостатки и преимущества по сравнению с классами.

Описание ключевых понятий:

Enum (перечисление) — это специальный тип данных в C#, который позволяет задавать переменную, принимающую фиксированный набор значений, представленных именованными константами. Перечисления делают код более читаемым и минимизируют вероятность ошибок.

Struct (структура) — это тип данных, который позволяет объединять различные элементы данных в одном агрегате. В отличие от классов, структуры являются типов значений, что означает, что при их передаче или присваивании создается копия данных. Структуры обычно используются для хранения небольшой группы логически связанных данных.

Nullable позволяет типам значений (таким как int, bool, DateTime и т.д.) принимать null, что обычно доступно только для ссылочных типов. Это полезно в ситуациях, когда необходимо указать, что данный тип может иметь значение или отсутствовать.

Перед выполнением лабораторной работы изучена следующая литература:

1. Изучена презентацию лектора курса: «Классы, структуры, конструкторы, модификаторы доступа в C# » (материалы доступны в "облаке" на Mail.ru и в Moodle КФУ).
2. Сайт Metanit.com
3. Справочник по C#. Корпорация Microsoft. http://msdn.microsoft.com/ru-ru/library/618ayhy6.aspx
4. Биллиг В.А. Основы программирования на C#. Интернет-университет информационных технологий. http://www.intuit.ru/studies/courses/2247/18/info
5. Павловская Т. Программирование на языке высокого уровня C#. http://www.intuit.ru/studies/courses/629/485/info
6. Руководство по программированию на C#. Корпорация Microsoft. http://msdn.microsoft.com/ru-ru/library/67ef8sbd.aspx
7. Корпорация Microsoft. C#. Спецификация языка. (Приложение A, Комментарии к документации).

Выполнены 4 задания, описанных в методических указания к выполнению лабораторных работ.

Задание 1. Структура комплексного числа

Для выполнения этого задания сделаем простую структуру из реальной части и мнимой

```
using System;
using System.Numerics;

namespace ComplexNumbers
{
    Ссылок: 61
    public struct Complex<T> where T : struct, INumber<T>
    {
        Ссылок: 27
        public T Real { get; set; }
        Ссылок: 28
        public T Imaginary { get; set; }

        Ссылок: 18
        public Complex(T real, T imaginary)
        {
            Real = real;
            Imaginary = imaginary;
        }

        Ссылок: 1
```

Для принятия чисел будем использовать Generic, где ограничим принимаемые параметры – только численные типы данных.

Сделаем метод для вычисления модуля комплексного числа

```
    {
        get
        {
            if (typeof(T) == typeof(double))
            {
                double realPart = Convert.ToDouble(Real);
                double imaginaryPart = Convert.ToDouble(Imaginary);
                double magnitude = Math.Sqrt(realPart * realPart + imaginaryPart * imaginaryPart);
                return (T)(object)magnitude;
            }
            else if (typeof(T) == typeof(float))
            {
                float realPart = Convert.ToSingle(Real);
                float imaginaryPart = Convert.ToSingle(Imaginary);
                float magnitude = (float)Math.Sqrt(realPart * realPart + imaginaryPart * imaginaryPart);
                return (T)(object)magnitude;
            }
            else
            {
                throw new NotSupportedException("Magnitude calculation is only supported for float and double.");
            }
        }
    }
    Ссылок: 0
    public T GetMagnitude()
    {
        return Magnitude;
    }
```

Перегрузим основным операции – сложение, вычитание, умножение, деление. Для сложения, вычитания и деления мы используем по 3 перегрузки. 1) Complex + Complex 2) Complex + Число 3) Число + Complex. Для деления будет два типа перегрузки 1) Complex / Число 2) Число / Complex

```
Ссылок: 1
Ссылок: 2
public static bool operator ==(Complex<T> a, Complex<T> b)
{
    return a.Real == b.Real && a.Imaginary == b.Imaginary;
}

Ссылок: 0
public static bool operator !=(Complex<T> a, Complex<T> b)
{
    return !(a == b);
}

Ссылок: 0
public override bool Equals(object obj)
{
    if (obj is Complex<T> other)
    {
        return this == other;
    }
    return false;
}
```

Перегрузим метод == и с ним != и Equals. В методе == будем сравнивать реальную и мнимую часть. А метод != будет работать в паре с Equals, который существует для того, чтобы сравнивать свои собственные структуры

Перегрузим метод ToString для корректного вывода

```
Ссылок: 0
public override string ToString()
{
    if (Imaginary > T.Zero)
    {
        return $"{Real} + {Imaginary}i";
    }
    else if (Imaginary < T.Zero)
    {
        return $"{Real} - {T.Abs(Imaginary)}i";
    }
    else
    {
        return $"{Real}";
    }
}
```

И сделаем два метода для преобразования числа из обычной(прямоугольной) формы в экспоненциальную

```
// Прямоугольная -> Экспоненциальная форма
Ссылок: 1
public (T Magnitude, T Argument) ToExponential()
{
    // Модуль r
    T magnitude;
    T argument;

    if (typeof(T) == typeof(double))
    {
        double realPart = Convert.ToDouble(Real);
        double imaginaryPart = Convert.ToDouble(Imaginary);
        magnitude = (T)(object)Math.Sqrt(realPart * realPart + imaginaryPart * imaginaryPart);
        argument = (T)(object)Math.Atan2(imaginaryPart, realPart);
    }
    else if (typeof(T) == typeof(float))
    {
        float realPart = Convert.ToSingle(Real);
        float imaginaryPart = Convert.ToSingle(Imaginary);
        magnitude = (T)(object)MathF.Sqrt(realPart * realPart + imaginaryPart * imaginaryPart);
        argument = (T)(object)MathF.Atan2(imaginaryPart, realPart);
    }
    else
    {
        throw new NotSupportedException("Conversion is only supported for float and double.");
    }
```

И обратное преобразование

```
// Экспоненциальная -> Прямоугольная форма
Ссылок: 1
public static Complex<T> FromExponential(T magnitude, T argument)
{
    if (typeof(T) == typeof(double))
    {
        double r = Convert.ToDouble(magnitude);
        double theta = Convert.ToDouble(argument);
        T real = (T)(object)(r * Math.Cos(theta));
        T imaginary = (T)(object)(r * Math.Sin(theta));
        return new Complex<T>(real, imaginary);
    }
    else if (typeof(T) == typeof(float))
    {
        float r = Convert.ToSingle(magnitude);
        float theta = Convert.ToSingle(argument);
        T real = (T)(object)(r * MathF.Cos(theta));
        T imaginary = (T)(object)(r * MathF.Sin(theta));
        return new Complex<T>(real, imaginary);
    }
    else
    {
        throw new NotSupportedException("Conversion is only supported for float and double.");
    }
}
```

Задание 2. Создание структуры для работы с трехмерными векторами.

Для выполнения этого задания также будем использовать Generic type. Сделаем конструктор для принятия трех чисел

```
public struct Vector3<T> where T : struct, IComparable, IFormattable, IConvertible, IComparable<T>, IEquatable<T>
{
    Ссылок: 19
    public T X { get; set; }
    Ссылок: 19
    public T Y { get; set; }
    Ссылок: 19
    public T Z { get; set; }

    Ссылок: 12
    public Vector3(T x, T y, T z)
    {
        X = x;
        Y = y;
        Z = z;
    }
```

Реализуем вычисление модуля отдельным свойством класса

```
public double Magnitude
{
    get
    {
        dynamic dx = X, dy = Y, dz = Z;
        return Math.Sqrt((double)(dx * dx + dy * dy + dz * dz));
    }
}

Ссылок: 0
public double GetMagnitude()
{
    return Magnitude;
}
```

Аналогично предыдущему заданию будем перегружать методы сложения вычитания используя три варианта использования

```
public static Vector3<T> operator +(Vector3<T> a, Vector3<T> b)
{
    dynamic dx = a.X, dy = a.Y, dz = a.Z;
    dynamic bx = b.X, by = b.Y, bz = b.Z;
    return new Vector3<T>(dx + bx, dy + by, dz + bz);
}

Ссылок: 0
public static Vector3<T> operator +(Vector3<T> a, T b)
{
    dynamic dx = a.X, dy = a.Y, dz = a.Z;
    dynamic scalar = b;
    return new Vector3<T>(dx + scalar, dy + scalar, dz + scalar);
}


public static Vector3<T> operator +(T a, Vector3<T> b)
{
    dynamic dx = b.X, dy = b.Y, dz = b.Z;
    dynamic scalar = a;
    return new Vector3<T>(dx + scalar, dy + scalar, dz + scalar);
}
```

В умножение и делении сначала отдельно рассмотрим операции вектора с числом

```
public static Vector3<T> operator *(Vector3<T> a, T scalar)
{
    dynamic dx = a.X, dy = a.Y, dz = a.Z;
    dynamic s = scalar;
    return new Vector3<T>(dx * s, dy * s, dz * s);
}

Ссылок: 0
public static Vector3<T> operator *(T a, Vector3<T> b)
{
    dynamic dx = b.X, dy = b.Y, dz = b.Z;
    dynamic s = a;
    return new Vector3<T>(dx * s, dy * s, dz * s);
}
```

```
public static Vector3<T> operator /(Vector3<T> a, T scalar)
{
    dynamic dx = a.X, dy = a.Y, dz = a.Z;
    dynamic s = scalar;
    return new Vector3<T>(dx / s, dy / s, dz / s);
}
Ссылок: 0
public static Vector3<T> operator /(T scalar, Vector3<T> b)
{
    dynamic dx = b.X, dy = b.Y, dz = b.Z;
    dynamic s = scalar;
    return new Vector3<T>(dx / s, dy / s, dz / s);
}
```

Перегрузим оператор сравнения

```
public static bool operator ==(Vector3<T> a, Vector3<T> b)
{
    return a.X.Equals(b.X) && a.Y.Equals(b.Y) && a.Z.Equals(b.Z);
}

Ссылок: 0
public static bool operator !=(Vector3<T> a, Vector3<T> b)
{
    return !(a == b);
}
```

Перегрузим метод ToString для корректного вывода вектора

```
public override string ToString()
{
    return $"({X}, {Y}, {Z})";
}
```

Отдельными методами напишем скалярное и векторное произведения векторов

```
Ссылок: 1
public static T Dot(Vector3<T> a, Vector3<T> b) // скалярное
{
    dynamic ax = a.X, ay = a.Y, az = a.Z;
    dynamic bx = b.X, by = b.Y, bz = b.Z;
    return ax * bx + ay * by + az * bz;
}

Ссылок: 1
public static Vector3<T> Cross(Vector3<T> a, Vector3<T> b) // векторное
{
    dynamic ax = a.X, ay = a.Y, az = a.Z;
    dynamic bx = b.X, by = b.Y, bz = b.Z;
    return new Vector3<T>(ay * bz - az * by, az * bx - ax * bz, ax * by - ay * bx);
}
```

Задание 3. Использование перечислений.

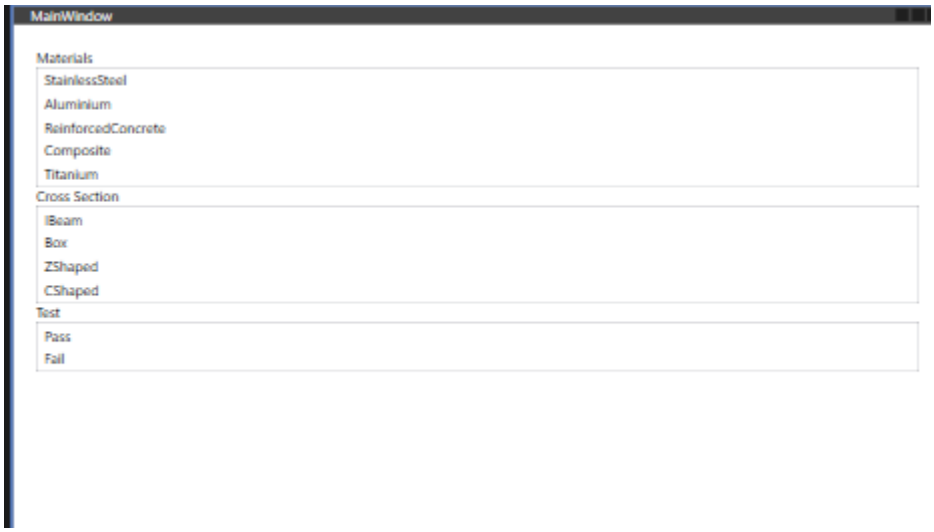
Сделаем Enum в пространстве StressTest

```
namespace StressTest
{
    /// <summary>
    /// Enumeration of girder material types
    /// </summary>
    Ссылок: 3
    public enum Material
    {
        StainlessSteel,
        Aluminium,
        ReinforcedConcrete,
        Composite,
        Titanium
    }

    /// <summary>
    /// Enumeration of girder cross-sections
    /// </summary>
    Ссылок: 3
    public enum CrossSection
    {
        IBeam,
        Box,
        ZShaped,
        CShaped
    }
```

Будем доставать значения из MainWindow все поля Enum’а

```
Ссылок: 3
private void MaterialList_SelectionChanged(object sender, SelectionChangedEventArgs e)
{
    Result.Content = ""; // Очистить перед добавлением нового текста

    if (MaterialList.SelectedItem is ListBoxItem materialItem)
    {
        Material SelMat = (Material)Enum.Parse(typeof(Material), materialItem.Content.ToString());
        Result.Content += $"{SelMat} ";
    }

    if (CrossList.SelectedItem is ListBoxItem crossSectionItem)
    {
        CrossSection SelCroSect = (CrossSection)Enum.Parse(typeof(CrossSection), crossSectionItem.Content.ToString());
        Result.Content += $"{SelCroSect} ";
    }

    if (TestList.SelectedItem is ListBoxItem testResultItem)
    {
        TestResult SelTes = (TestResult)Enum.Parse(typeof(TestResult), testResultItem.Content.ToString());
        Result.Content += $"{SelTes} ";
    }
}
```

Сделаем пользовательский интерфейс



Задание 4. Использование структур

Для выполнения добавим метод в StressTest.cs

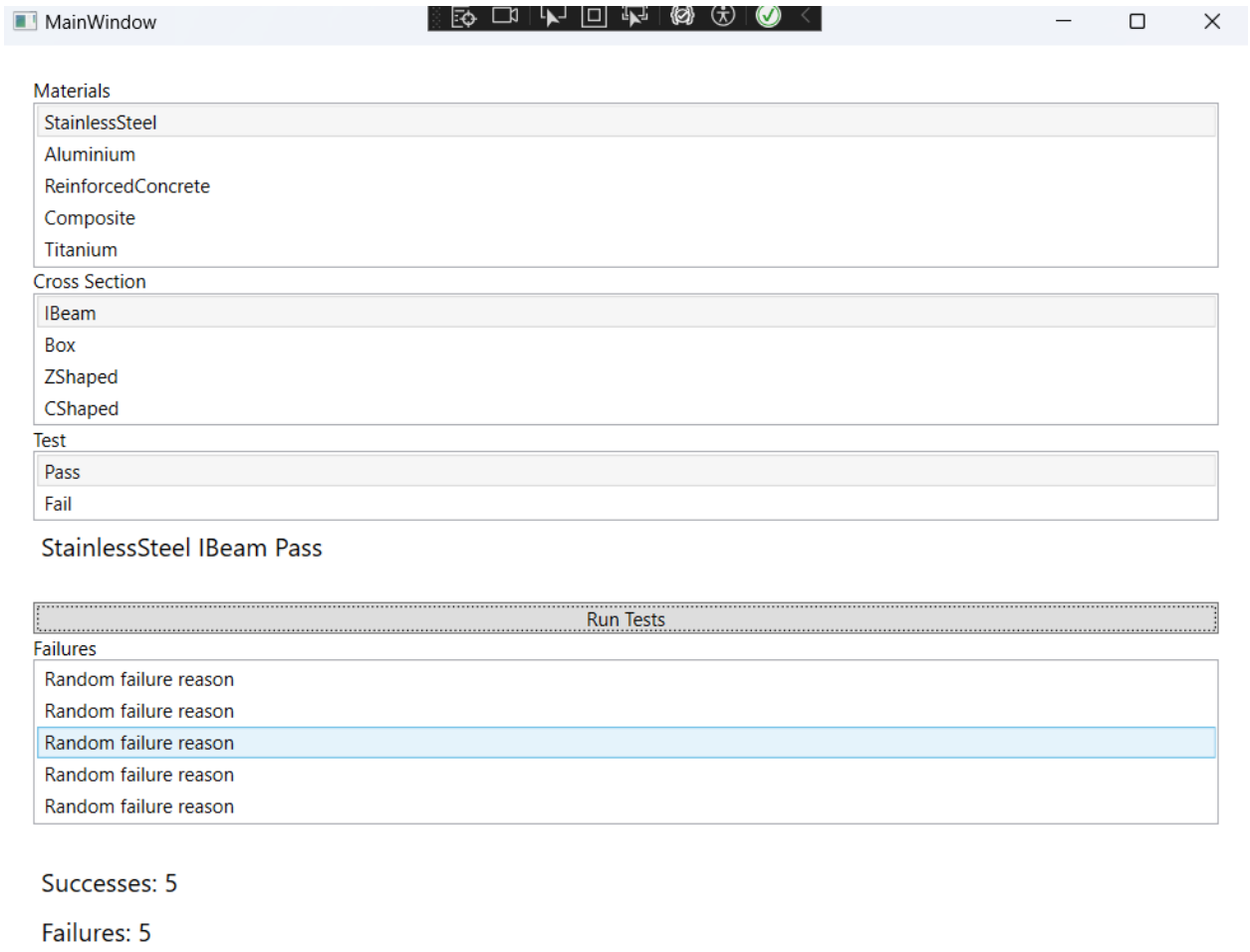
```
public struct TestCaseResult
{
    /// <summary>
    /// Test result (enumeration type)
    /// </summary>
    public TestResult Result;

    /// <summary>
    /// Description of reason for failure
    /// </summary>
    public string ReasonForFailure;

    // Constructor to initialize the fields
    Ссылок: 0
    public TestCaseResult(TestResult result, string reasonForFailure)
    {
        Result = result;
        ReasonForFailure = reasonForFailure;
    }
}
```

Для этого добавим метод RunTests, которое частично будет реализовать подсчет правильности. Будет стоять выбор рандомного числа четное будет удачей, нечетное неудачей

```
Ссылок: 1
public static void RunTests(TestCaseResult[] results, ListBox reasonsList, Label successesLabel
{
    // Clear the reasonsList list
    reasonsList.Items.Clear();

    // Initialize the results array
    results = new TestCaseResult[10];

    // Fill the results array with data
    for (int i = 0; i < results.Length; i++)
    {
        results[i] = TestManager.GenerateResult();
    }

    // Display the TestCaseResult data
    int passCount = 0;
    int failCount = 0;

    foreach (var result in results)
    {
        if (result.Result == TestResult.Pass)
        {
            passCount++;
        }
        else if (result.Result == TestResult.Fail)
        {
            failCount++;
            reasonsList.Items.Add(result.ReasonForFailure);
        }
    }
```

```
public static class TestManager
{
    private static Random random = new Random();

    Ссылок: 1
    public static TestCaseResult GenerateResult()
    {
        // Simulate a random test result
        TestResult result = random.Next(2) == 0 ? TestResult.Pass : TestResult.Fail;
        string reasonForFailure = result == TestResult.Fail ? "Random failure reason" : "";

        return new TestCaseResult(result, reasonForFailure);
    }
}
```

В пользовательском интерфейсе сделаем подсчёт



Ответы на вопросы:

1. Недостатки и преимущества использования структур по сравнению с классами

Преимущества структур:

- **Легковесность**: Структуры обычно занимают меньше памяти, чем классы, так как они хранятся в стеке, а не в куче.
- **Производительность**: Работа со структурами может быть быстрее, так как они не требуют сборки мусора.
- **Копирование**: При передаче структуры в метод или при присваивании ее другой переменной создается копия, что может быть полезно для небольших объектов.
- **Потокобезопасность**: Структуры могут быть полезны в многопоточных приложениях, так как каждая копия структуры независима.

Недостатки структур:

- **Не поддерживают наследование**: Структуры не могут наследоваться от других структур или классов, и сами не могут быть базовыми для других типов.
- **Не поддерживают деструкторы**: Структуры не могут иметь деструкторы, так как они не управляют ресурсами, которые нужно освобождать.

- **Не поддерживают ссылочную семантику**: Структуры всегда копируются при передаче, что может быть неэффективно для больших объектов.
- **Не поддерживают null**: Структуры не могут быть null, что может быть ограничением в некоторых сценариях.

2. Изменения в использовании конструкторов структур в C# 8, C# 10, C# 11

C# 8:

- **Конструкторы без параметров**: В C# 8 структуры могут иметь конструкторы без параметров, но они должны быть явно определены.
- **Инициализация полей**: Поля структуры могут быть инициализированы в конструкторе без параметров.

C# 10:

- **Конструкторы без параметров по умолчанию**: В C# 10 структуры могут иметь конструкторы без параметров по умолчанию, которые инициализируют все поля значениями по умолчанию.
- **Инициализация полей в объявлении**: Поля структуры могут быть инициализированы в объявлении, даже если конструктор не определен.

C# 11:

- **Конструкторы без параметров по умолчанию**: В C# 11 конструкторы без параметров по умолчанию могут быть использованы для инициализации полей структуры.
- **Инициализация полей в объявлении**: Поля структуры могут быть инициализированы в объявлении, даже если конструктор не определен.

3. Перегрузка операторов

Как осуществляется перегрузка операторов?

Перегрузка операторов позволяет определить поведение операторов (+, -, *, /, ==, !=, и т.д.) для пользовательских типов данных. Для перегрузки оператора используется ключевое слово operator вместе с именем оператора.

Какие операции можно перегрузить?

В C# можно перегрузить следующие операторы:

- Арифметические операторы: +, -, *, /, %
- Операторы сравнения: ==, !=, <, >, <=, >=
- Логические операторы: &&, ||
- Битовые операторы: &, |, ^, ~, <<, >>
- Унарные операторы: +, -, !, ~, ++, --
- Операторы приведения: (T)x

4. Преимущество использования перечислений

Преимущества перечислений:

- **Типобезопасность**: Перечисления обеспечивают типобезопасность, так как позволяют использовать только определенные значения.
- **Читаемость кода**: Перечисления делают код более читаемым и понятным, так как вместо магических чисел используются осмысленные имена.

- **Упрощение поддержки**: Перечисления упрощают поддержку кода, так как добавление новых значений требует изменения только в одном месте.
- **Автоматическая документация**: Перечисления автоматически документируют возможные значения, что упрощает понимание кода.

5. Nullable типы

Что такое Nullable тип?

Nullable тип — это тип данных, который может принимать как обычное значение, так и значение null. В C# это достигается с помощью оператора ?. Например, int? — это Nullable тип для целочисленного значения.

Зачем нужен Nullable тип?

- **Обработка отсутствующих значений**: Nullable типы позволяют обрабатывать случаи, когда значение может отсутствовать.
- **Совместимость с базой данных**: Nullable типы упрощают работу с базами данных, где значения могут быть null.
- **Улучшение безопасности**: Nullable типы помогают избежать ошибок, связанных с NullReferenceException, так как компилятор может предупреждать о возможных null значениях.

Представлены 4 проекта, реализованных в Visual Studio Community 2022. Проекты представлены преподавателю в электронной форме, продемонстрирована их работоспособность, разъяснены детали программного кода.