Теория кодирования

Лабораторная работа №1: Кодирование по методам Шеннона-Фано, Хаффмана и арифметическому методу.

Задание: написать программу, которая кодирует (сжимает) данные по методу Шенона-Фано, Хаффмана и по арифметическому методу. В качестве данных должно выступать ФИО студента. По первым двум методам необходимо дополнительно вывести перечень вероятностей букв в текстовом поле. Для каждого метода нужно определить % сжатия данных, как отношение объёма данных ПОСЛЕ кодирования, к объёму данных ДО кодирования.

* Кодирование методом Шеннона-Фано

Пусть имеется строка из 6 букв. Строка является массивом байт, которые содержат код каждого элемента из ASCII. Таким образом, объём будет равен как байт \* длину строки = 6 байт. Алгоритм Шеннона-Фано позволяет сжать информацию без потерь, оперируя вероятностями появления отдельных элементов.

Первый шаг - определяем таблицу вероятностей букв строки (без повторения символов):

P('a1') = 0,25

P('a2') = 0,125

P('a3') = 0,25

P('a4') = 0,25

P('a5') = 0,0625

P('a6') = 0,0625

Второй шаг – расставляем вероятности в порядке убывания:

P('a1') = 0,25

P('a3') = 0,25

P('a4') = 0,25

P('a2') = 0,125

P('a5') = 0,0625

P('a6') = 0,0625

Теперь начинаем итерационно разбивать буквы на группы, у которых суммы вероятностей равны или примерно одинаковы (разность сумм между группами минимальная). Далее одной группе присваиваем по левую сторону присваиваем 0, по правую сторону - 1, и начинаем повторный процесс разбиения уже для каждой новой группы похожим образом.

Таблица 1 – Первая итерация групп

|  |  |  |
| --- | --- | --- |
| Группы | Сумма вероятностей | Код |
| P(a1) = 0,25  P(a3) = 0,25 | ∑ = 0.5 | 0 |
| P(a4) = 0,25  P(a2) = 0,125  P(a5) = 0,0625  P(a6) = 0,0625 | ∑ = 0.5 | 1 |

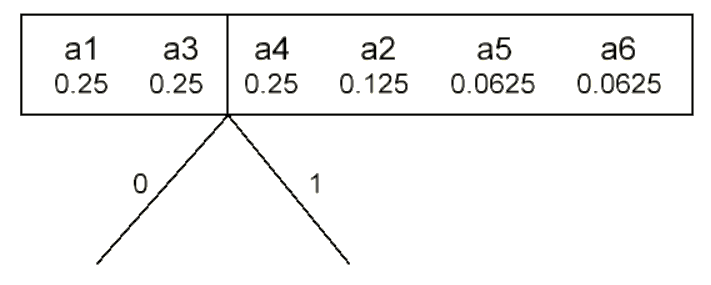


Рисунок 1 – Дерево первой итерации

Делаем вторую итерацию, для которой разбиваем группу слева и справа ещё на две группы:

Таблица 2 – Вторая итерация групп

|  |  |  |
| --- | --- | --- |
| Группы | Сумма вероятностей | Код |
| P(a1) = 0,25 | 0.25 | 00 |
| P(a3) = 0,25 | 0.25 | 01 |
| P(a4) = 0,25 | 0.25 | 10 |
| P(a2) = 0,125  P(a5) = 0,0625  P(a6) = 0,0625 | ∑ = 0.25 | - |

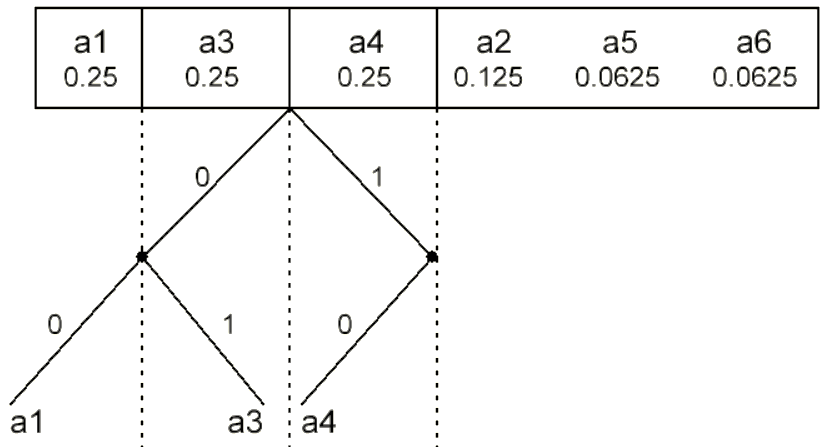


Рисунок 2 – Дерево второй итерации

Таблица 3 – Третья итерация групп

|  |  |  |
| --- | --- | --- |
| Группы | Сумма вероятностей | Код |
| P(a1) = 0,25 | - | 00 |
| P(a3) = 0,25 | - | 01 |
| P(a4) = 0,25 | - | 10 |
| P(a2) = 0,125 | 0.125 | 110 |
| P(a5) = 0,0625  P(a6) = 0,0625 | ∑ = 0.125 | - |

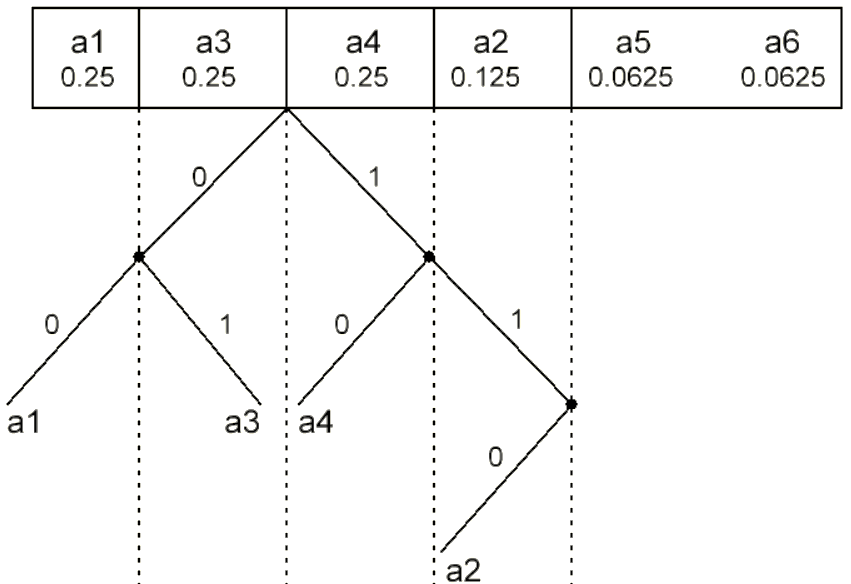


Рисунок 3 – Дерево третьей итерации

Таблица 4 – Четвёртая итерация групп

|  |  |  |
| --- | --- | --- |
| Группы | Сумма вероятностей | Код |
| P(a1) = 0,25 | - | 00 |
| P(a3) = 0,25 | - | 01 |
| P(a4) = 0,25 | - | 10 |
| P(a2) = 0,125 | - | 110 |
| P(a5) = 0,0625 | 0.0625 | 1110 |
| P(a6) = 0,0625 | 0.0625 | 1111 |

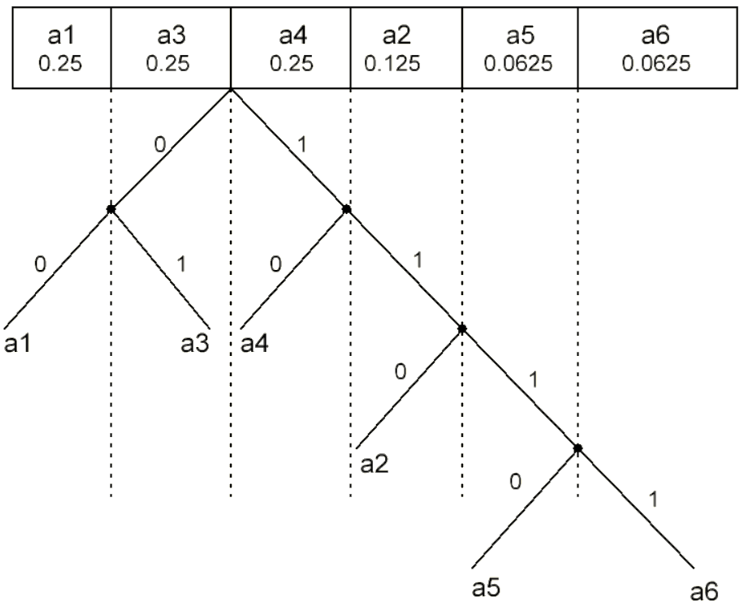


Рисунок 4 – Дерево четвёртой итерации

* Кодирование по Хаффману

Кодирование по Хаффману похоже на кодирование по методу Шеннона-Фано, за исключением того, что дерево растёт ветвями вверх.

Возьмём пример с пятью вероятностями каждой буквы в слове:

P('a1') = 0.5

P('a2') = 0,1

P('a3') = 0,1

P('a4') = 0,05

P('a5') = 0,25

Следующим шагом расставляем вероятности их в порядке убывания:

P('a1') = 0,5

P('a5') = 0,25

P('a2') = 0,1

P('a3') = 0,1

P('a4') = 0,05

Далее, объединяем два символа с наименьшими вероятностями в группу. В этой группе, символу с большей вероятностью присваиваем код 1, а с меньшей вероятностью – 0. В случае одинаковых вероятностей, ставим коды в любом порядке.

|  |  |  |
| --- | --- | --- |
| Группы | Код | Сумма |
| P('a1') = 0,5  P('a5') = 0,25  P('a2') = 0,1 | - | - |
| P('a3') = 0,1  P('a4') = 0,05 | 1  0 | 0.15 |

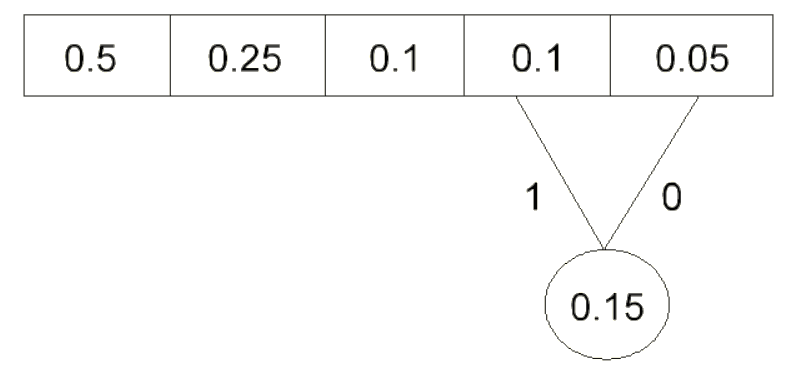


Рисунок 5 – Дерево первой итерации

Затем, считаем сумму этих двух вероятностей в группе, и повторяем действия. **Важное замечание** – объединять можно только те вероятности, которые равны или примерно соизмеримы (как в методе Шенона-Фано). В противном случае, складывается новая группа вероятностей, и далее объединяется со следующей.

|  |  |  |
| --- | --- | --- |
| Группы | Код | Сумма |
| P('a1') = 0,5  P('a5') = 0,25 | - | - |
| P('a2') = 0,1  P('a3') + P('a4') = 0.15 | 0  1 | 0.25 |

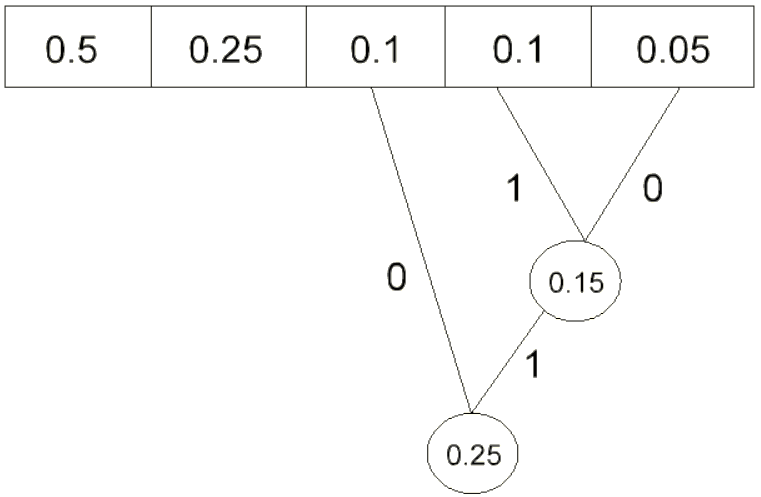


Рисунок 6 – Дерево второй итерации

Третья итерация:

|  |  |  |
| --- | --- | --- |
| Группы | Код | Сумма |
| P('a1') = 0,5 | - | - |
| P('a5') = 0,25  P('a2') + P('a3') + P('a4') = 0.25 | 0  1 | 0.25 |

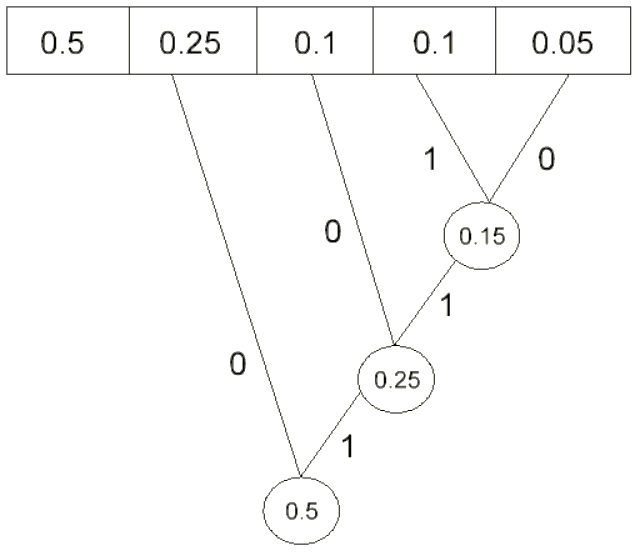


Рисунок 7 – Дерево третьей итерации

|  |  |  |
| --- | --- | --- |
| Группы | Код | Сумма |
| P('a1') = 0,5  P('a5') + P('a2') + P('a3') + P('a4') = 0.5 | 0  1 | 1 |

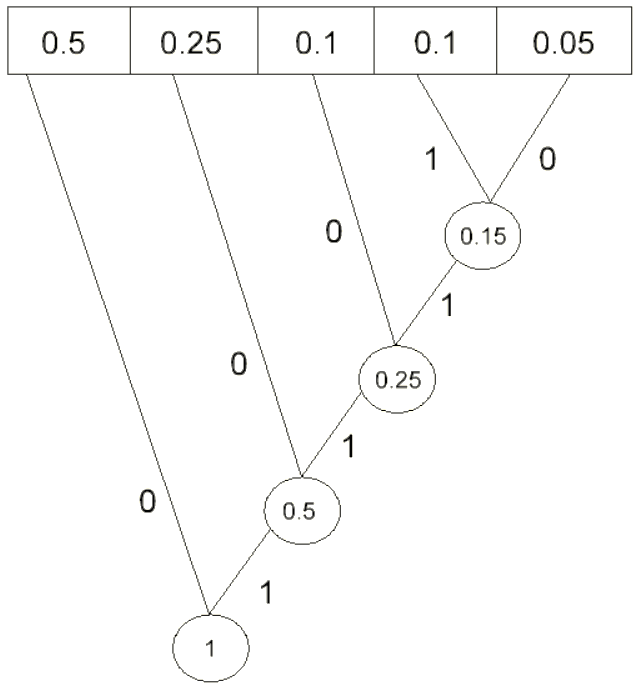


Рисунок 8 – Дерево четвёртой итерации

Далее, считая снизу вверх, запишем кодовые последовательности для каждого символа:

P('a1') = 0.5 → 0

P('a5') = 0,25 → 10

P('a2') = 0,1 → 110

P('a3') = 0,1 → 1111

P('a4') = 0,05 → 1110

Счёт в дереве всегда идёт от корня к ветвям.

1 Подготовка к работе

1.1 Создание проекта в VS2022

Лабораторная работа подготовлена для работы на языке C# с библиотеками Windows Forms. Однако если у вас есть опыт работы в библиотеке WPF C# или в языке Java, можете работать с ними. Однако, общая концепция работы (окна и алгоритмы) должны соответствовать методическим указаниям.

Откройте Visual Studio 2022 и выберете создать проект, и выберете проект «приложение Windows Forms (.NET Framework)».

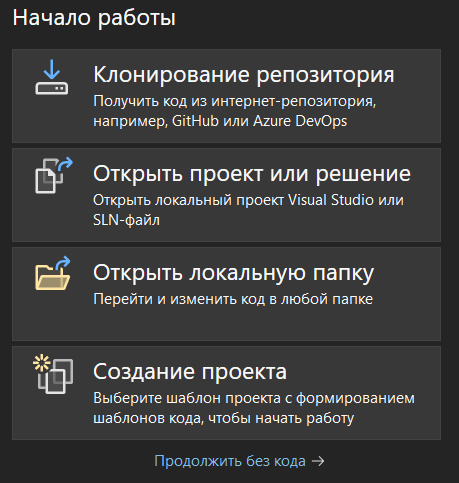


Рисунок 1 – Создание проекта

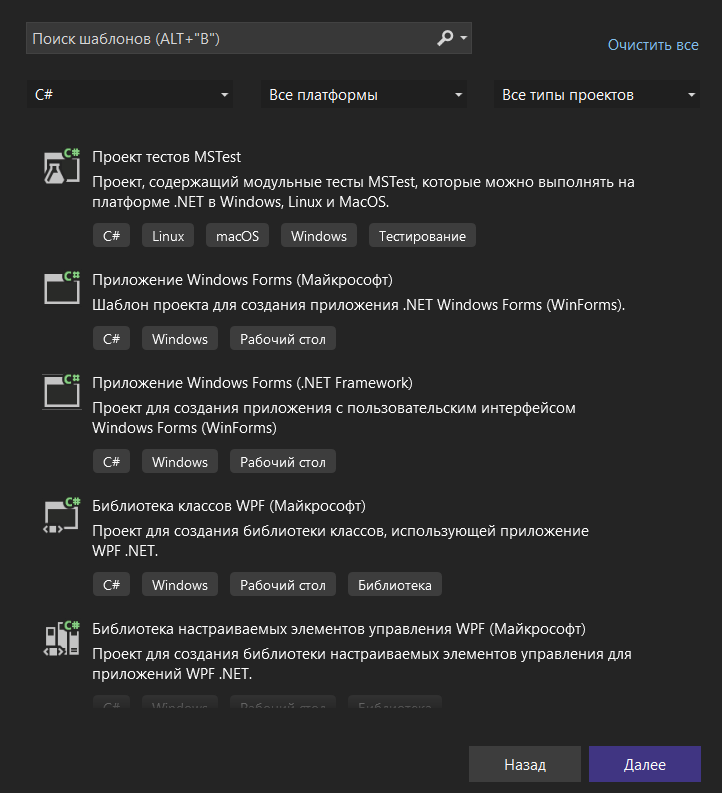


Рисунок 2 – Выбор типа проекта

Имя проекта должно содержать номер лабораторной работы, имя и номер группы.

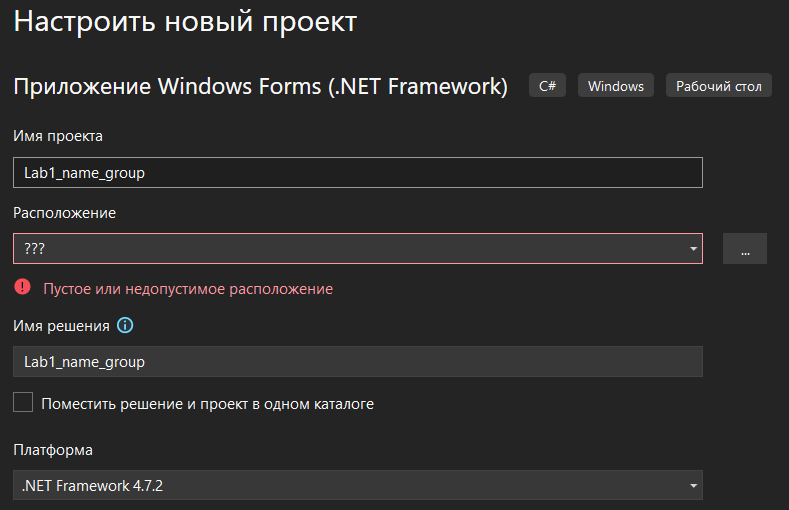


Рисунок 3 – Задание имени и пути файлов

Далее, после завершения настройки появится рабочая область конструктора формы.

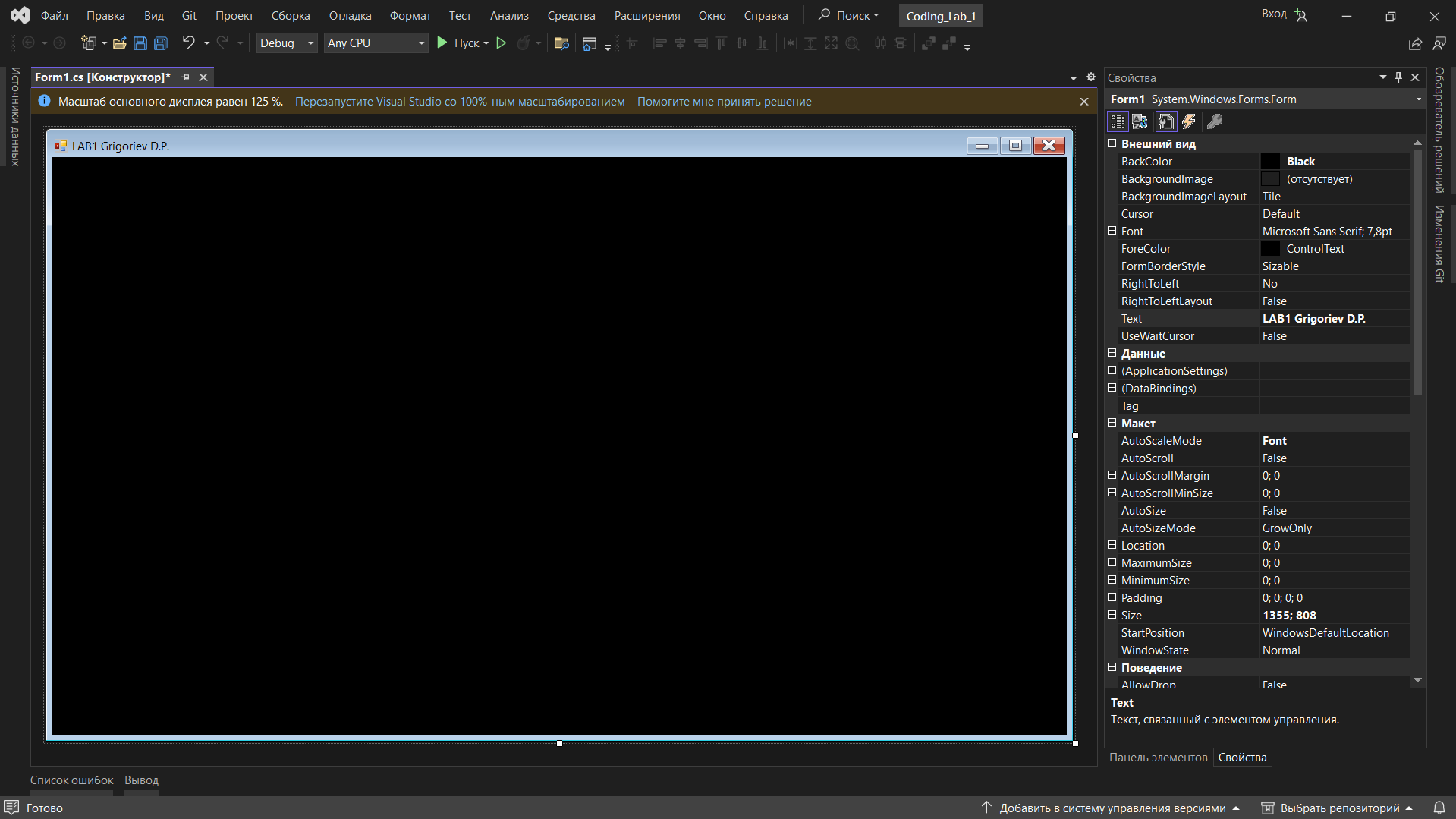


Рисунок 4 – Окно работы

Если боковая панель элементов будет отсутствовать, нужно её вывести через меню Вид – Панель элементов.

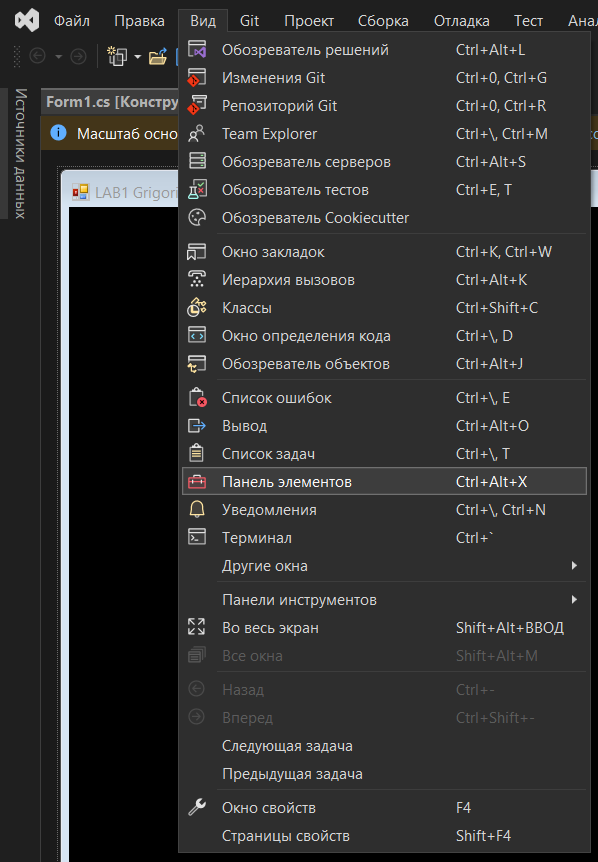


Рисунок 5 – Включение панели элементов и прочих окон

Далее, в панели элементов вам понадобится элемент TabControl, который поможет организовать вкладки с окнами в вашей форме. Для ввода значений примените TextBox, а для информативных пометок используйте Label. Для обработки действий пользователя можно применить любой элемент, но чаще всего используются кнопки button.

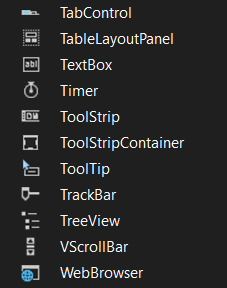


Рисунок 6 – Элемент TabControl и textbox

Перенесите необходимые элементы на поле и разместите их схожим образом (рисунок 7). Для добавления вкладок справа вверху используйте кнопку «добавить вкладку» (рисунок 8). Для настройки параметров окон и вкладок, кликните правой кнопкой мыши на требуемый элемент и нажмите «свойства». Откроется боковая панель со свойствами, где можно регулировать цвет фона, шрифт и прочие элементы (рисунок 9).

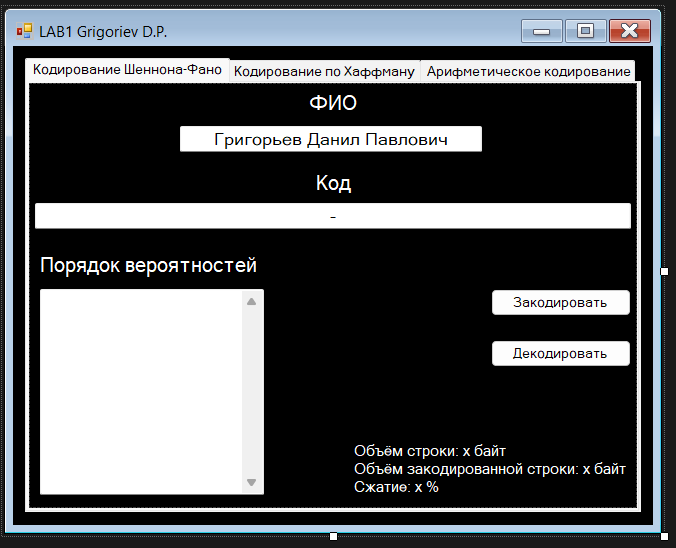


Рисунок 7 – Пример оформления окна для лабораторной работы

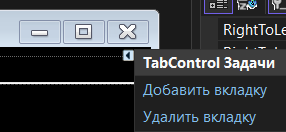


Рисунок 8 – Изменение количества кладок

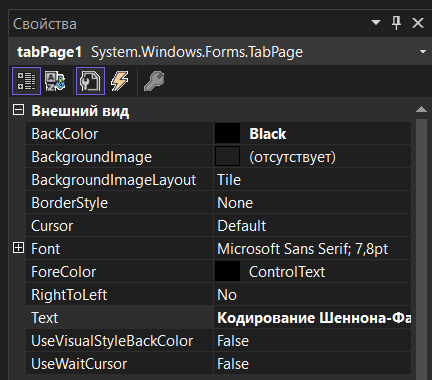


Рисунок 9 – Свойства страницы tabPage1

Для того, чтобы перейти к коду формы, нажмите по ней правой мыши и выберете перейти к коду (горячая клавиша F7).

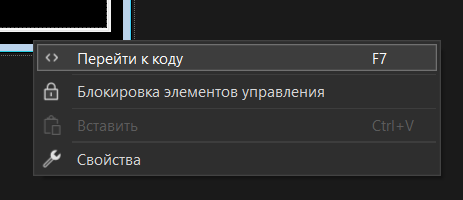


Рисунок 10 – Переход к коду формы

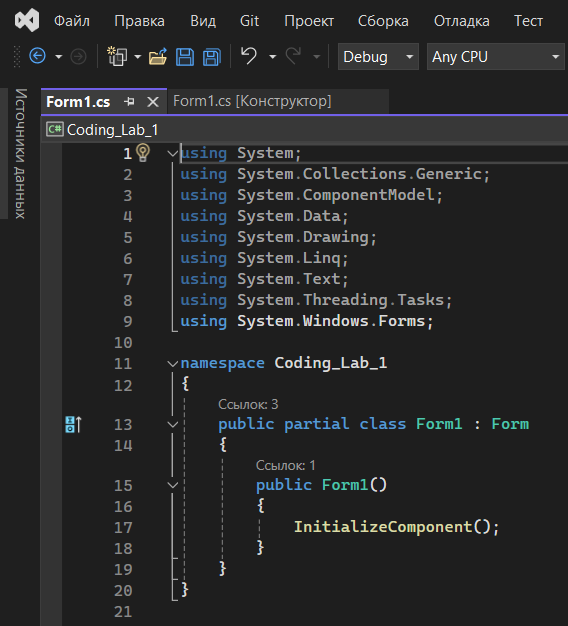


Рисунок 11 – Стартовый код пустой формы

Чтобы задать действие у кнопки в момент нажатия, нажмите по ней правой кнопкой мыши и выберете свойства (рисунок 12). Далее нажмите на значок молнии, которая переключит вас на панель задания событий текущего элемента. В графе Click напишите имя метода, который будет вызываться при нажатии кнопки (рисунок 13). После введения имени, конструктор автоматически создаст метода в коде формы, и перенесёт вас к нему.

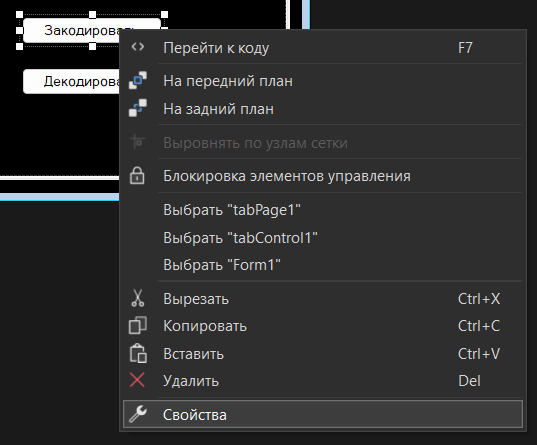


Рисунок 12 – Выбор свойств кнопки

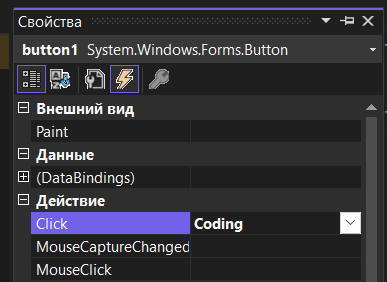


Рисунок 13 – Задание методы по событию нажатия кнопки

1.2 Базовые понятия C#

1.2.1 Структура кода C#

Язык программирования C# построен на библиотеках «NET Framework» и «NET Core» от Microsoft, и содержит весь базовый функционал, необходимый для работы программ на платформе Windows. Библиотеки, для работы с графическими интерфейсами, реализованы в пакетах Windows Forms и WPF. Оба они подняты на библиотеках WinAPI. Windows Forms создаёт файл «Designer.cs» в котором прописаны все элементы графики, а в WPF применяется код .XAML, который описывает расположения и размеры элементов. Сами листинги хранятся в файлах с расширением \*.cs (c-sharp).

Код, написанный на C#, является безопасным, поскольку он не содержит явных указателей на память, что исключает разного рода утечки памяти. Также платформа имеет встроенный сборщик мусора (garbage collection), который очищает неиспользуемые ссылки на объекты в памяти. Вся библиотека NET отлажена и содержит обработку исключений до того, как неверный код будет запущен процессором (не доходя до runtime error). Однако есть способ обойти эти ограничения и использовать указатели, подобно языку СИ. Для этого возле блоков кода надо расположить ключевые слова «unsafe», что допускается только в крайних случаях.

Напомним иерархию ООП в языках программирования. Любой код на ООП содержит несколько базовых компонентов:

* пространство имён (namespace);
* класс (class);
* методы (methods);
* свойства (properties);
* глобальные и локальные операнды (переменные).

Первым делом задаётся пространство имён, в рамках которого реализуются классы. Далее, классы содержат методы работы, которые характеризуют уже механику работы кода. Методы могут обращается как ко внутренним методам и свойствам, так и к внешним (если это не противоречит модификаторам).

Рассмотрим архитектуру кода на примере автомобиля:

namespace Car

{

class Engine

{

void EngON()

{

//to do…

}

void EngOFF()

{

//to do…

}

void Diagnostic(Date date)

{

//to do…

}

}

class Cabin

{

void SetCabin(int chairs, double temperature, bool Light)

{

//to do…

}

void CabinLock(bool State)

{

//to do…

}

void StreetLight(bool state)

{

//to do…

}

}

class Trunk

{

void Open()

{

//to do…

}

void Close()

{

//to do…

}

void Lock(bool State)

{

//to do…

}

}

}

Итак, пространство имён «автомобиль» имеет три класса – двигатель, кабина, багажник. Каждый класс имеет конкретный набор методов, в соответствии со своим названием (набор характеристик и способов управления). Так, класс двигателя имеет методы включения и выключения двигателя, и метод диагностики двигателя. Класс кабины содержит методы установки количества сидений, температуры в кабине, света, блокировку дверей и метод запуска уличных фар. Класс багажника содержит методы открытия / закрытия багажника, и блокировки его замка.

Для использования этих классов необходимо подключить код через пространство имён в заголовке программы:

using car;

Классы, описанные выше, являются динамическими, и вызываются в вашей программе через экземпляры (объекты), которые представляют собой буфер – копию класса, содержащий все изменения, в процессе работы кода. Имя экземпляра может быть любым, например:

Engine myEng = new Engine(); // создание экземпляра myEng

myEng.EngOn(); //вызов метода класса Engine

Если класс содержит конструктор (метод, имеющий такое же название, как и класс), и в его аргументах необходимо ввести какие-нибудь параметры, то их надо ввести при инициализации экземпляра (например, максимальная скорость автомобиля).

Engine myEng = new Engine(10); // создание экземпляра myEng с V=10

myEng.EngOn(); //вызов метода класса Engine

Если перед ключевым словом class добавить static, то вы дадите классу свойство статичности, и по нему уже нельзя будет создавать экземпляры.

static class Engine

{…}

Методы такого класса вызываются напрямую (через оператор точка « . »). Свойства и глобальные операнды его также меняются напрямую. Такой способ используется, если вы хотите содержать в классе изменённые данные, который регистрируются разными классами в одном или разных пространствах имён. Такой подход похож на глобальную переменную, только доступ к ней происходит из любого места программы без надобности передачи и создания лишних экземпляров.

Engine.EngOn(); //вызов метода статичного класса

Методы в рамках одного класса могут вызываться друг у друга без создания экземпляров.

void SetCabin(int chairs, double temperature, bool Light)

{

//to do…

StreetLight(true);

//to do...

}

1.2.2 Модификаторы доступа

Методы из класса пользователь может вызывать, если они не содержат модификатор доступа, или содержат модификатор public. Такие модификаторы ограничивают методы на вызовы, которые могут быть использованы пользователем. Например, пользователю доступны только методы включения и выключения двигателя, но ему закрыт доступ на вызов метода диагностики двигателя:

public void EngON() //модификатор публичного доступа

{

//to do…

}

void EngOFF() //автоматический public

{

//to do…

}

//доступ только в рамках класса Engine

private void Diagnostic(Date date)

{

//to do…

}

Это необходимо, чтобы пользователь не смог затронуть важные методы класса, которые могу повредить программу. Методы private используются только в рамках работы самого класса и его методов. Модификаторы могут применятся ко всем участкам кода, будь то класс, метод, свойство или операнд. Всего различают 5 частых модификаторов доступа:

* public - код в любой сборке может получить доступ к этому типу или члену;
* private - только код, объявленный в том же class или struct может получить доступ к этому члену;
* protection - только код в том же class или в производном class может получить доступ к этому типу или члену;
* internal - доступ к этому типу или члену может получить только код в одной сборке;
* file - доступ к типу или члену может получить только код в одном файле.

1.2.3 Типы данных

Типы данных простых операндов не отличаются от операндов на языке СИ и С++. Операнды определяются тем же способом: тип\_данных имя\_операнда = занчение:

int X = 5; //Целочисленный тип

double Y = 5.25; //Тип числа с плавающей запятой 2х

string str = “это строка”; //Строковый тип

char symbol = ‘L’; //Байтовый символьный тип

byte b1 = 0xFF; //Байтовый без знаковый тип

UInt16 w1 = 0xFFFF; //Двух байтовый без знаковый тип

UInt32 dw1 = 0xFFAAFFAA; //Четырёх байтовый без знаковый тип

short w2 = -0x10; //Двух байтовый целочисленный тип

Перечислительные типы данных:

int[] arr1 = new int[15]; //Вариант массива на 15 ячеек

int[] arr1 = new int[3] {3,4,5}; //Вариант массива c данными

int[] arr1 = {3,4,5}; //Вариант массива c данными без //указания размера

int[,] arr2 = new int[1,2]; //Матрица 1х2

Перечислительный динамический список:

List<int> arr1 = new List<int>(); //Динамический список типа int

List<T> arr1 = new List<T>(); //Динамический список типа T generic

arr1[2] = 55; //Запись числа в ячейку

Y = arr1[1]; //Чтение ячейки

arr1.Length; //Получение длины массива

Динамический список имеет ряд преимуществ перед обычными массивами. Разница в них состоит лишь в том, что лист является аналогом вектора на С++, который умеет накапливать ячейки с данными и удалять их, без заранее установленного размера массива (списка). Обозначим распространённые методы работы и свойства списка:

List<int> arr1 = new List<int>();

arr1.Add(25); //Добавление объекта

arr1.RemoveAt(10); //Удаление объекта по его индексу

arr1.Remove(1); //Удаление указанного объекта

arr1.Count; //Получение размера листа

Остальные операции по перебору и чтении ячеек такие же, как и в обычном массиве.

Стоит отметить особый тип списка T – это тип generic, который указывается при инициализации списка, если разработчик допускает любой тип данных листа. Такой подход требуется, например, для реализации библиотеки очереди или дерева, когда ячейками могут выступать данные любого типа. Поскольку алгоритм работы кода должен быть одинаковым, то чтобы не дублировать весь код библиотеки под каждый тип данных, применяют generic, который является как бы операндом для типов. Таким образом, вместо буквы T в дальнейшем может быть любой тип данных, даже структура, класс или кортеж.

Тип словаря:

Dictionary<int, string> dict1 = new Dictionary<int, string>();

Словарь – это тип данных, который содержит в одной ячейке пару: ключ и значение. Словарь отличается от списка тем, что индексы в данном случае как таковые отсутствуют, ибо вместо них используются ключи, которые могут быть любым типом данных. В данном примере ключ имеет тип «int», а значение данных «string». Это необходимо для быстрой идентификации данных, например, когда поиск указывается по ключу, получаемому из хеш-функции для паролей и т.д.

dict1[5] = “word”; //Изменение данных по ключу

dict.ContainsKey(5); //Проверка наличия ключа

dict.ContainsValue("progger"); //Проверка наличия данных

dict.Count(); //Получение размера словаря

Ключ для каждого значения уникальный. Чтение и изменение данных происходит как с обычным массивом, только в качестве индекса используется ключ. В целом, словарь является по сути своей списком, и имеет ряд тех же методов и свойств.

Выше описанные методы списка и словаря могут быть преобразованы в массив методом ToArray(). Также и наоборот, массив может быть переделан в любой из динамических типов данных: ToList<T>(); ToDictionary<T,T>().

Тип кортеж:

(int, double) LR = (25, 4.23); //Кортеж без имён операндов

(int X, double Y) LR = (25, 4.23); //Кортеж с именами операндов

Кортеж по своей сути является кратким способом объявления списка данных с разными типами данных. Элементов в кортеже может быть сколь угодным. Доступ к значениям в первом случае осуществляется через ключевое слово item:

LR.Item1 = 25;

LR.Item2 = 4.23;

Во втором случае доступ осуществляется через имена:

LR.X = 25;

LR.Y = 4.23;

Кортеж является таким же типом, как и остальные типы данных, поэтому их можно использовать как в списках, так и в словаре:

var limitsLookup = new Dictionary<int, (int Min, int Max)>()

{

[2] = (4, 10),

[4] = (10, 20),

[6] = (0, 23) //[key] = кортеж с данными

};

Тип структуры – это тип данных, который может содержать внутри себя несколько операндов разного типа.

Struct DatContainer

{

int X = 5;

string exmpl = “Пример структуры”;

bool state = false;

}

Остальная инициализация и работа со структурой схожа с обычным классом.

DatContainer DC = new DatContainer();

int Z = DC.X;

DC.state = true;

Для начальной инициализации данных используется конструктор по умолчанию, аргументами которого являются все элементы структуры в порядке их приведения:

DatContainer DC = new DatContainer(5, “Пример структуры”, false);

Можно обращается к каждому элементу отдельно:

DatContainer DC = new DatContainer {X = 5, exmpl = “Пример структуры”, state = false};

Допускается реализация своего конструктора структуры, который записывает в элементы определённый (или весь) перечень операндов:

Struct DatContainer

{

private int X = 5;

private string exmpl = “Пример структуры”;

private bool state = false;

public DatContainer(int p1 = 5, string p2 = “test”)

{

X = p1;

exmpl = p2;

//пропущен элемент state

}

}

//инициализация по прописанному конструктору

DatContainer DC = new DatContainer(5, “Пример структуры”);

Стоит отметить важное замечание. На языке C# нельзя изменять структуру явно по элементам, если структура использовалась при инициализации, например, списка:

//Инициализация списка с типом нашей структуры

List<DC> myDcList = new List<DC>();

//Добавление ячейки с нашей структурой

myDCList.Add(new DC {X = 10, exmpl = “test”, state = false});

//Изменение элемента структуры в ячейки списка [ошибка]

myDCList[0].X = 50; //ОШИБКА

Ошибка возникает ввиду того, что последним выражением мы работаем не с элементом структуры, а с копией элемента X. Поэтому изменив его, мы никак не повлияем на реальное значение в структуре. Для изменения элементов применяется сторонний экземпляр-буфер, который записывает все значения структуры, меняет элемент, и возвращает целиком структуру:

//Инициализация списка с типом нашей структуры

List<DC> myDcList = new List<DC>();

//Добавление ячейки с нашей структурой

myDCList.Add(new DC {X = 10, exmpl = “test”, state = false});

//Делаем буфер структуры

DatContainer buffer = new DatContainer();

//Копируем структуру

buffer = myDCList[0];

//Изменение элемента структуры в буфере

buffer.X = 50;

//Возврат изменённой структуры

myDCList[0] = buffer;

Перечислительный тип с экземплярами класса.

Вместо инициализации списка по структуре, как было показано выше, обычно используют инициализацию по экземплярам класса (каждая ячейка списка – это отдельный экземпляр класса). Например, допустим у нас есть класс Engine. Добавим в класс несколько операндов и пару свойств:

class Engine

{

int speed = 0;

public double temperature {get; set;}

void EngON()

{

//to do…

}

void EngOFF()

{

//to do…

}

void Diagnostic(Date date)

{

//to do…

}

}

Теперь мы можем определить список:

//Инициализация списка с типом нашего класса

List<Engine> myDcList = new List<Engine>();

//Добавление ячейки с нашим экземпляром класса

myDCList.Add(new Engine());

//Изменение элементов класса

myDCList.speed = 100;

1.2.4 Свойства

Ещё один важный компонент в ООП – это свойство. По своей природе, свойство схоже с обычной глобальной переменной, которое создаётся вне методов, и доступ к ней есть как у всех методов класса, так и у самого класса (при вызове через экземпляры и т.д.).

int speed = 10;

void Diagnostic(Date date)

{

speed ++;

//to do…

}

Свойство же является элементом, предоставляющий гибкий механизм для чтения, записи или вычисления значения поля данных. Свойства отображаются как открытые элементы данных, но они реализуются как специальные методы, называемые методами доступа. Эта функция позволяет легко получать доступ к данным и по-прежнему способствовать обеспечению безопасности и гибкости данных. Объявляется свойство как метод:

public int speed { get; set; }

Оператор «get» позволяет пользователю прочитать значение «speed», а оператор «set» позволяет записать число в «speed». Рассмотрим следующие способы инициализации свойств:

* запись числа в свойство: public int speed { set; }
* чтение числа из свойства: public int speed { get; }
* запись числа и запрет чтения внешним кодом:

public int speed { private get; set; }

* чтение числа и запрет записи внешним кодом:

public int speed { get; private set; }

* задание первоначального значения:

public int speed { get; set; } = 5;

* задание свойства через конструктор класса («set» не требуется):

class Engine

{

public Engine(int maxSpeed) => speed = maxSpeed;

public int speed { get; }

}

Оператор « => » является основой лямда-выражения, когда код пишется после стрелки одной строкой (или множеством строк, но в фигурных скобках). Аналогично можно было бы записать:

class Engine

{

public Engine(int maxSpeed)

{

speed = maxSpeed;

}

public int speed { get; }

}

1.2.5 События

События являются по сути прерываниями в рамках работы программы. Например, события могут вызываться в момент нажатия какой-либо кнопки на интерфейсе, или прихода сообщений по Ethernet. Для передачи данных между дочерней и родительской формой можно также применить события. Для примера сделаем две формы – родительскую и дочернюю. Пусть в дочерней форме глобально прописано событие и его делегат:

public event CMD CMD\_Select;

public delegate void CMD(int cmd, int data);

В делегате указываем перечень переменных, которые будут передаваться в обработчик события. Типы данных и количество переменных может быть любым, или отсутствовать вовсе. Причём, имя делегата должно совпадать с именем события.

После этого в родительской форме создаём экземпляр дочерней формы, подписываемся на её событие, и открываем форму:

Form form1 = new Form(); //создаём экземпляр

form1.CMD\_Select += (cmd, data) => //обработка события

{

//to do…

};

form1.Show(); //показ формы

Оператор += означает подписку обработчика на событие, и их может быть сколько угодно. Для отписки используется оператор -=. Обработчик события написан с помощью лямбда выражения. Перед стрелкой в круглых скобках указываются данные, которые отправляются в делегат дочерней формы. Имена могут отличатся, в данном случае, это не критично. Эти имена используются внутри обработчика для дальнейшей работы. После стрелки идёт блок выполняемого кода, помещённый в фигурных скобках.

Теперь, когда в дочерней форме вызовут событие через Invoke:

CMD\_Select.Invoke(10, 50);

то вызовется обработчик в родительской форме, и получит данные, которые указаны в аргументах Invoke.

1.2.6 Потоки

Любая программа представляет собой один процесс. В рамках программы реализуется минимум один поток (основная программа). Далее может быть сколько угодно потоков, в зависимости от необходимости алгоритма. Под потоком подразумевается блок независимого кода (подпрограмма), которая запускается параллельно основной программе. Это полезно в случаях, если необходимо распараллелить вычисления, или отдельные подпрограммы так, чтобы они не мешали друг другу. В некоторых случаях в потоках удобно реализовывать вечные циклы, задержки и прочие фоновые алгоритмы.

Потоки бывают асинхронными и синхронными. Синхронный поток, блокирует другие потоки, пока тот не завершится. Это может повлиять на зависание интерфейса, когда, например, запускается поток математических преобразований, а в это время пользователь хочет отменить операцию вычисления. В это случае кнопки нажать не получится чисто физически, поскольку выполнение кода будет в другом потоке. Асинхронный же поток не влияет на работу основных потоков, даже если он зависнет. Поэтому пользовательский интерфейс всегда обособляется от сложных вычислительных модулей асинхронностью, чтобы иметь возможность остановить работу программы. Иногда при применении потоковых вычислениях стоит задуматься о регистрации их завершения, если речь идёт о громоздком вычислении разных формул, которые должны будут свестись к одному результату. Тогда программа должна ожидать, когда все потоки завершатся.

Инициализация потока начинается с объявления экземпляра:

Thread t1 = new Thread(new ThreadStart(Method\_1));

В аргументах указывается делегат ThreadStart, которому передаётся ссылка (имя) запускаемого метода Method\_1. Таким образом, мы запускаем делегат, который будет выполнять в потоке метод Method\_1. Сам метод описывается также, как и остальные:

void Method\_1() {…}

Запуск потока и остановка происходит следующими методами:

t1.Start();

t1.Abort();

После остановки потока, запустить его заново не получится. Для паузы потока используется метод Join(). Перед закрытием программы, все побочные потоки необходимо отключать, иначе они будут висеть в памяти постоянно.

Чтобы не прописывать отдельный метод, как показано выше, можно использовать лямбда выражение, и прописать строки кода сразу при инициализации потока:

Thread t1 = new Thread(() =>

{

Thread.Sleep(10);

});

Бывают случаи, когда нужно передать данные из дочернего потока в основной, например, в форму (вывести сообщение на textbox). Поэтому применяется следующий алгоритм работы с помощью лямбда выражений:

BeginInvoke(new Action(delegate ()

{

textbox1.Text = “Поток вызван”;

}));

BeginInvoke - это метод, который позволяет запустить указанный делегат асинхронно в отдельном потоке. Обычно используется, когда нужно выполнить задачу параллельно без блокировки основного потока. Action() - инкапсулирует метод, который не имеет параметров и не возвращает значений. Этот делегат можно использовать для передачи метода в качестве параметра без явного объявления пользовательского делегата. В данном случае, запускается лямбда выражение. delegate() - это тип, который безопасно инкапсулирует метод, схожий с указателем функции в C и C++. В отличие от указателей делегаты объектно-ориентированы, и типобезопасны.

Ещё один вариант передачи данных, без указания делегата:

BeginInvoke(new Action(() =>

{

textbox1.Text = “Поток вызван”;

}));

Помимо базового класса Thread имеется класс Task, который написан поверх Thread. Он имеет более гибкий инструмент к реализации потоков на C#.

Task task1 = new Task(() =>

{

     Thread.Sleep(1000);

  });

task1.Start();  // запускаем задачу асинхронно

task1.RunSynchronously(); // запускаем задачу синхронно

task1.Wait();   // ожидаем выполнения задачи

В рамках задания асинхронного метода MyLoop(), необходимо указать ключевое слово async перед типом Task, и далее через ключевое слово await вызвать запуск задачи в виде лямбда выражения:

public static async Task MyLoop()

{

await Task.Run(() =>

{

for (int ctr = 0; ctr <= 1000000; ctr++);

} );

}

Для больших затрат, задачи можно размножить, если использовать массив Task[]. После выполнения метода, поток автоматически завершается.

1.2.7 Таймер

Таймер предоставляет возможность вызова определённых методов через определённое время (интервал), задаваемое в миллисекундах. Таймеры часто применяют для выслеживания таймаута или запуска расписания команд. Чаще всего, по истечении интервала, вызывается событие Tick, по которому задаёт соответствующий метод обработки таймера.

Timer tim1 = new Timer(); //создание экземпляра

tim1.Interval = 1000; //задание времени срабатывания Tick

tim1.Tick += timer1\_handler; //подписка обработчика на событие

tim1.Start(); //запуск таймера

Обработчик представляет собой обычный метод:

void timer1\_handler()

{

//to do...

}

Обработчик можно вызвать асинхронно, по уже знакомым задачам:

tim1.Tick += async ( sender, e ) => await timer1\_handler ();

Теперь метод примет вид:

private static Task timer1\_handler ()

{

Console.WriteLine("timer is worked!" );

}

Таймер, в отличие от потоков, можно останавливать и запускать в любое время любое количество раз. За отслеживание работы таймера, служит свойство Enabled.

1.2.8 Парадигмы проектирования кода на ООП

Проектирование любого ПО состоит из пяти принципов:

* Абстракция - это процесс выделения общих характеристик и функциональности объектов или системы, игнорируя детали реализации. Пример продемонстрирован в п. 1.2.1, когда код реализован по типу автомобиля. Применена явная абстракция между автомобилем, и кодом, который описывает автомобиль. Параметрами служат свойства класса, которые инициализируются заранее при вызове конструктора класса. Напомним, что конструктором класса является метод, имеющий такое же имя, как и класс. Поэтому при создании экземпляра, после оператора new следует имя класса, которое заменяется методом - конструктором:

class Engine

{

public int speed {get; set;}

public Engine(int maxSpeed)

{

speed = maxSpeed;

}

}

* Инкапсуляция — это принцип, согласно которому внутреннее устройство сущностей (классов) нужно объединять в специальной «оболочке» - капсуле, и скрывать от вмешательств извне. Доступ к объектам возможен через специальные открытые методы, а напрямую обратиться к их содержимому нельзя. Такой подход описан в п. 1.2.2, когда используются модификаторы доступа private.
* Наследование – это концепция, согласно которой одни классы, называемые родительскими, могут лежать в основе других — дочерних. При этом дочерние (производные) классы перенимают свойства и поведение своего родителя.

Пример:

public class A //базовый класс

{

public void Method1()

{

// Method implementation.

}

}

public class B : A //Класс B наследует A

{

//to do...

}

public class Example //Некая класс

{

public static void Main() //Программа

{

B b = new (); //Экземпляр класса B

b.Method1(); //Вызов метода базового класса

}

}

В некоторых случаях методы можно переопределить в производных классах, для этого в базовом классе напротив переопределяемого метода необходимо написать ключевое слово virtual, а у переопределения override:

public class A //базовый класс

{

public virtual void Method1()

{

// Method implementation.

}

}

public class B : A //Класс B наследует A

{

public override void Method1()

{

//Переопределение метода базового класса

}

}

public class Example //Некая класс

{

public static void Main() //Программа

{

B b = new (); //Экземпляр класса B

b.Method1(); //Вызов метода класса B

}

}

В некоторых случаях требуется насильно переопределить метод базового класса, если у него стоит модификатор abstract:

public abstract class A

{

public abstract void Method1();

}

public class B : A

{

public void Method3() //Ошибка, отсутствует //переопределение Method1

{

// to do...

}

}

* Полиморфизм - один из ключевых принципов ООП, позволяющий разным сущностям (классам и методам) выполнять одни и те же действия, при этом неважно, как они устроены внутри и чем различаются. Полиморфизм подразделяется на параметрический и мнимый (ad-hoc). Первый применяется тогда, когда у наследованных классов, один метод выполняется для каждого класса отдельно, в зависимости от конкретного экземпляра.

Пример:

Мнимый полиморфизм является перегрузкой методов, когда в коде существует несколько методов с одинаковыми именами, но с разными типами данных (или количеством) аргументов.

Пример:

int sum(int a, int b) {…}

int sum(string a, string b) {…}

int sum(int a, int b, int c) {…}

int sum(double a, double b) {…}

* Интерфейс - это структура программы или синтаксиса, определяющая отношение с объектами, объединёнными только некоторым поведением. Другими словами, в интерфейсе содержится перечень методов (без реализации), которые может принять любой класс, и реализовать их код по своему.

Пример:

interface ImyInt\_face //Создаём интерфейс

{

void Method1(); //Объявляем методы

}

class ImplementClass : ImyInt\_face //Наследуем интерфейс

{

void ImyInt\_face.Method1()

{

//Реализуем код метода интерфейса

}

static void Main() //Основная программа

{

//Создаём экземпляр интерфейса по классу, где он реализован

ImyInt\_face interface1 = new ImplementClass ();

interface1.Method1(); //Вызываем метод

}

}