МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РФ
Государственное образовательное учреждение высшего образования
«ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ СИСТЕМ УПРАВЛЕНИЯ И РАДИОЭЛЕКТРОНИКИ» (ТУСУР)

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ (C#)

Учебно – методическое пособие по курсу «Объектно-ориентированное программирование» для выполнения практических, лабораторных работ и проведения самостоятельной работы для студентов ВУЗа

Томск
2018

Пособие составлено в соответствии с тематикой практических, лабораторных работ и самостоятельной работы по курсу «Объектно-ориентированное программирование». Пособие содержит темы и содержание практических и лабораторных работ, методические указания к их проведению.

Для преподавателей, аспирантов, студентов и магистрантов.

СОСТАВИТЕЛЬ: Е.А. Шельмина

СОДЕРЖАНИЕ

Раздел 1. Практические работы

Практическая работа №1

Введение в объектно-ориентированное программирование

По мере развития вычислительной техники возникали разные методики программирования. На каждом этапе создавался новый подход, который помогал программистам справляться с растущим усложнением программ. Первые программы создавались посредством переключателей на панели компьютера. Вполне очевидно, что подобного рода способ подходил только для небольших программ. Затем программы стали писать на языке машинных команд. С изобретением ассемблера (язык низкого уровня) стали появляться сравнительно длинные программы. Настоящим прорывом в программировании стало создание первого языка программирования высокого уровня - Фортран в 1950 году. С этого времени появилась возможность писать программы до нескольких тысяч строк длинной. Однако увеличение объема программ привело к тому, что код больших программ становился практически нечитаемым, а зачастую даже неуправляемым. Избавление от подобного рода проблем неструктурного программирования пришло с изобретением в начале 60-х годов таких языков структурного программирования как Алгол, С и Паскаль. Применяя эти принципы программирования, появилась возможность создания и поддержки программ в несколько десятков тысяч строк. Несомненно, что структурное программирование принесло выдающиеся результаты, но, как оказалось, даже эти принципы программирования становятся несостоятельными, когда программа достигает определенной длины. Для написания более сложных программ потребовался новый подход к программированию. В итоге были разработаны принципиально новые методы программирования и на свет появилось объектно-ориентированное программирование (ООП), в котором нашли отражение лучшие идеи структурного программирования в сочетании с абсолютно новыми концепциями, позволяющими оптимально организовать программу.

Все языки ООП, основаны на трех основополагающих концепциях, называемых инкапсуляцией, полиморфизмом и наследованием.

Инкапсуляция - это механизм, который объединяет данные и код, манипулирующий с этими данными, а также защищает и то, и другое от внешнего вмешательства или неправильного использования. В объектно-ориентированном программировании код и данные могут быть объединены вместе. Когда код и данные объединяются таким способом, создается объект. Другими словами, объект - это то, что поддерживает инкапсуляцию.

Полиморфизм - это свойство, которое позволяет одно и то же имя использовать для решения двух или более схожих, но технически разных задач. Целью полиморфизма, применительно к ООП, является использование одного имени для задания общих для класса действий. Выполнение каждого конкретного действия будет определяться типом данных.

Наследование - это процесс, посредством которого один объект может приобретать свойства другого. Точнее, объект может наследовать основные свойства другого объекта и добавлять к ним черты, характерные только для него. Наследование позволяет поддерживать концепцию иерархии классов.

Язык программирования C#

Язык программирования C# — это объектно-ориентированный язык со строгой типизацией, позволяющий разработчикам создавать различные безопасные и надежные приложения, работающие на платформе .NET Framework. C# можно использовать для создания клиентских приложений Windows, XML-веб-служб, распределенных компонентов, приложений клиент-сервер, приложений баз данных и т. д. Visual C# предоставляет развитый редактор кода, удобные конструкторы пользовательского

интерфейса, интегрированный отладчик и многие другие средства, которые упрощают разработку приложений на языке C# для платформы .NET Framework.

Структура языка C#

Любой язык программирования подобен естественному языку человека. В языках программирования также можно выделить алфавит, слова и предложения. Но язык программирования в отличии от естественного языка гораздо проще и подчиняется жестким правилам.

Алфавит языка — это набор символов, который допустим в данном языке. В алфавит языка C# входят:

- прописные и строчные буквы
- символ подчеркивания (_)
- цифры
- специальные знаки: ' " , { } | [ ] ( ) + - / \ % ? ! . ; :< = > & * ~ ^
- неотображаемые символы, которые используются для отделения лексем друг от друга (это пробелы, табуляция, переход на новую строку).

Из отдельных символов алфавита языка строятся более крупные блоки программы: лексемы, директивы препроцессора и комментарии.

Лексема — это последовательность из одного или нескольких символов, представляющая определенный смысл. Существует несколько видов лексем:

- идентификаторы (имена объектов);
- ключевые (зарезервированные, служебные) слова;
- знаки операций;
- разделители.

Идентификатор — это последовательность букв, цифр и символов подчеркивания. Идентификаторы используются для обозначения имен объектов, используемых в программе. Всё, что применяется в программе, имеет имя. Имена (названия) имеют константы, переменные, методы, классы и т. д. Имя не может начинаться с цифры. Длина имени произвольная.

Ключевые слова — это служебные слова, которые зарезервированы в языке, их можно использовать только по прямому назначению (например, for — это заголовок оператора цикла и ничего более), т. е. зарезервированные слова нельзя использовать в качестве имен переменных пользователя.

Знаки операций — это один или несколько символов, определяющих действие над операндами. Внутри знака операции не может быть пробелов (пробел — это всегда разделитель). Например, в выражении x+y знак + означает операцию сложения, а x и y являются операндами.

Константы — это величины, которые неизменны. В C# допустимы целые, вещественные, символьные, строковые, логические константы и константа null. Компилятор определяет тип константы по ее внешнему виду.

Управляющая последовательность представляет собой определенный символ, перед которым стоит обратная косая черта "\". Такие последовательности необходимы для обозначения символов, не имеющих графического представления (например, \n — переход на новую строку), а также для символов, которые имеют специальное значение в символьных и строковых константах, например, апостроф. Ниже приведены некоторые значения управляющих последовательностей:

\a - звуковой сигнал
\b - возврат на одну позицию
\f - подача страницы (для перехода к началу следующей странице)
\n - переход на новую строку
\r - возврат каретки
\t - горизонтальная табуляция
\v - вертикальная табуляция

\0 - нуль-символ
\' - апостроф (одинарная кавычка)
\" - двойная кавычка
\\ - обратная косая черта

Комментарии — это фрагменты программы, которые не обрабатываются компилятором, а используются для пояснения текста программы. В языке C# имеются те же виды комментариев, что и в C++.

Типы данных

Тип данных характеризует основные свойства данных: возможные действия с данными, их внутреннее представление, диапазон допустимых значений, размер. Все типы данных условно можно подразделить на простые (обычно это встроенные типы) и структурированные (структуры, классы). В языке C# все простые типы и структуры относятся к значащим типам (в переменных таких типов непосредственно хранятся значения данных соответствующих типов), а классы — к ссылочным типам. Память для переменных значащих типов выделяется в стеке, для ссылочных объектов — в «общей куче» (heap).

Рассмотрим подробнее простые типы, к которым относятся целые типы, вещественные, символьные, логические и некоторые другие.

Целые типы

Целые типы данных предназначены для хранения целых чисел. Все целые типы можно подразделить на знаковые и беззнаковые. В таблице 1 приведены все знаковые целые типы и их основные характеристики.

Таблица 1

Характеристика целых знаковых типов языка C#

Тип	Размер, байт	Диапазон допустимых значений
sbyte	1	-128 … 127
short	2	-32768 … 32767
int	4	-2147483648 … 2147483647
long	8	-9223372036854775808 … 9223372036854775807

В таблице 2 приведены все беззнаковые целые типы данных.

Таблица 2

Характеристика целых беззнаковых типов языка C#

Тип	Размер, байт	Диапазон допустимых значений
byte	1	0 … 255
ushort	2	0 … 65535
uint	4	0 … 4294967295
ulong	8	0 … 18446744073709551615

Для целых типов допустимы следующие группы операций:
арифметические операции;
побитовые операции;
операции сравнения;
инкремент и декремент.

В языке C# возможно автоматическое преобразование типов от менее мощного к более мощному типу, но не наоборот. Например:

int i = 3;

```
short k = 2;
i = k; // это допустимо
k = i; // не верно - будет сообщение об ошибке
k = (short) i; //верно - выполняем явное преобразование
```

Вещественные типы

Для работы с действительными числами в языке C# имеется два вещественных типа: float (одинарная точность) и double (удвоенная точность). Для вещественных типов допустимы следующие группы операций:

арифметические операции;

операции сравнения;

инкремент и декремент.

Тип decimal

Этот тип специально создан для выполнения денежных расчётов. Обеспечивает точность до 29 десятичных цифр. Диапазон возможных значений (по модулю) от 10^{-28} до 10^{28}. Под переменные этого типа отводится 16 байт.

Символьный тип

Для хранения отдельных символов используется тип char. Так как в языке C# для кодирования символов используется Unicode, то под каждый символ отводится 2 байта. Пример объявления переменной этого типа: char c.

Логический (булевый) тип

Для работы с логическими величинами используется тип bool. Допустимы два значения: true (истина) и false (ложь). Для данных логического типа допустимы операции проверки на равенство и неравенство. Следует иметь в виду, что для булевого типа в C# запрещены какие-либо преобразования в целые типы.

Константы и переменные

Константа — это величина, которая при выполнении программы остаётся неизменной.

Переменная — это ячейка памяти для временного хранения данных. Предполагается, что в процессе выполнения программы значения переменных могут изменяться.

Описание и инициализация переменных

Прежде чем использовать в программе какую-то переменную, надо обязательно дать её описание, то есть сказать, какое имя имеет переменная и каков её тип. Вначале указывается тип переменной, а затем её имя. Например:

```
int k; // это переменная целого типа int
double x; // это переменная вещественного типа удвоенной точности
```

Если имеется несколько переменных одного типа, то допускается их описание через запятую в одном операторе, например: double a, b, c;

Инициализация и использование констант

Все константы вне зависимости от типа данных можно подразделить на две категории: именованные константы и константы, которые не имеют собственного имени.

Например:

25 — константа целого типа;

3.14 — вещественная константа;

'A' — символьная константа.

Все три приведённые здесь константы не имеют имени, они заданы своим внешним представлением и используются в программе непосредственно, например так:

```
int k=25; // переменная k инициализирована константой — целым числом 25.
```

В ряде случаев константе удобнее дать имя и использовать её далее по имени. Это делает текст программы более наглядным.

В языке C# определить именованную константу очень просто. Для этого используют ключевое слово const, например:

```
const double PI=3.14; // здесь PI — константа
double t;
t=PI * 2;
```

Операции

Операции — это действия над данными. Данные, участвующие в операции, часто называют операндами. В качестве операнда может выступать константа, переменная или вызов какого-нибудь метода. Для каждой операции используется соответствующий ей знак операции, состоящий из одного или нескольких символов. В результате выполнения операций всегда получается какое-то значение, представляющее собой результат выполнения операции.

В языке C# существует большое количество разнообразных операций. Их можно классифицировать по различным признакам, например: по количеству операндов, по назначению. В зависимости от количества операндов в C# есть унарные, бинарные и тернарная операция.

По назначению операции можно сгруппировать следующим образом: арифметические, операции сравнения, логические, побитовые, специальные.

Арифметические операции

Это наиболее часто используемые операции. Их смысл близок к тому, каким он известен из курса математики. Итак, перечислим их:

+ сложение

- вычитание

* умножение

/ деление (обычное)

/ целочисленное деление

% вычисление остатка при целочисленном делении (применима только для целочисленных операндов).

В C# один знак / означает две разные операции. Если один или оба операнда — вещественные, то выполняется обычное деление, если оба операнда — целые, то выполняется деление нацело и результат будет целого типа.

Операция инкремент

Данная операция записывается следующим образом: ++, применяется для увеличения на единицу значения переменной, например: A++. Исходное значение переменной A увеличивается на 1, и полученный результат сохраняется в переменной A. По полученному результату эта операция соответствует следующему выражению: A=A+1.

Операция инкремента применима именно для переменной, но не для константы или выражения. Для операции инкремент допустимы две формы записи:

префиксная — например, ++A

постфиксная — например, A++.

При префиксной форме записи делается увеличение переменной на 1 и затем используется новое значение этой переменной. В постфиксной форме записи также переменная увеличивается на 1, но в текущем выражении используется старое значение переменной.

Операция декремент

Записывается как --, применяется для уменьшения на единицу значения переменной, например: A--. В остальном аналогична операции инкремента.

Операции отношения

Применяются для сравнения переменных числовых или символьных типов. Результатом выполнения таких операций является либо истина (true), либо ложь (false). Перечислим эти операции:

< — меньше;

<= — меньше или равно;

== — проверка на равенство (пишется два знака «равно» без пробелов);

!= — проверка на неравенство;
> — больше;
>= — больше или равно.

Логические операции

Логические операции тесно связаны с операциями отношения и используются для построения сложных логических выражений. Имеются следующие логические операции:

логическое отрицание (НЕ) - ! !
логическое умножение (И) - &&
логическое сложение (ИЛИ) - ||

Математические функции

Как и в большинстве языков программирования, в языке C# имеется большой набор математических функций. Все математические функции реализованы в виде статических методов в классе Math, который в свою очередь определён в области имён System. Для того, чтобы в программе на C# использовать математические функции, необходимо подключать область имён System, а при вызове метода, реализующего ту или иную математическую функцию, явно указывать название класса Math.

Наиболее распространенные методы класса Math:

Abs(x) - вычисляет модуль (абсолютное значение) числа x;
Acos(x) - функция арккосинуса;
Asin(x) - функция арксинуса;
Atan(x) - функция арктангенса;
Cos(x) - функция косинуса;
Exp(x) - вычисляет значение (экспоненциальная функция);
Log(x) - возвращает значение натурального логарифма (ln x);
Log10(x) - возвращает значение десятичного логарифма ( );
Max(a, b) - возвращает максимум из двух чисел a и b;
Min(a, b) - возвращает минимум из двух чисел a и b;
Pow(x, a) - возводит число x в степень a;
Sin(x) - функция синуса;
Sqrt(x) - возвращает положительное значение квадратного корня;
Tan(x) - функция тангенса.

Ключевое слово var

Начиная с версии C# 3.0 в язык было добавлено ключевое слово var, которое позволяет создавать переменные без явного указания типа данных. Тип данных такой переменной определяет компилятор по контексту инициализации.

```
static void Main(string[] args)
{
  var number = 5; // number будет типа int
  var text ="some text"; // text будет типа string
  var number2 = 0.5; // number2 будет типа double
}
```

var сохраняет принцип строгой типизации в C#. Это означает, что после того, как для переменной уже был определен тип, в нее нельзя записать данные другого типа:

```
static void Main (string[] args)
{
  var number = 5;
  number = "some text"; // ошибка, number определен как int
}
```

Ключевое слово var имеет ограничения по его использованию - var не может быть в качестве: поля класса, аргумента функции, возвращаемого типа функции, переменной, которой присваивается null.

Ссылочные типы

В языке C# кроме рассмотренных выше типов, ещё есть ссылочные типы. Из базовых типов к ссылочным относятся object и string. Тип object является базовым для всех остальных типов данных. Типу string соответствует строка символов Unicode.

Пример использования типа string:

```
static void Main(string[] args)
{
  string hello = "Hello!";
  Console.WriteLine(hello); }
```

Операторы

Оператор — это законченное предложение, записанное на каком-либо языке программирования. Каждый оператор в программе на C# обязательно заканчивается символом ; (точка с запятой). Операторы условно можно подразделить на две категории: исполняемые — с их помощью реализуется алгоритм решаемой задачи, и описательные, необходимые для определения типов пользователя и объявления объектов программы, например, переменных.

Все исполняемые операторы можно разбить на две группы: простые и структурированные. К простым операторам можно отнести оператор присваивания, пустой оператор, операторы переходов (goto, break, continue, return), оператор-выражение, вызов метода как отдельного оператора.

Структурированные операторы — это сложные (составные) операторы, которые могут объединять в себе другие операторы. К этой категории относятся операторы ветвления if, выбора switch и операторы циклов (for, while, do, foreach).

Теперь более подробно рассмотрим простые операторы.

Оператор присваивания

Назначение этого оператора — присвоить новое значение какой-либо переменной. В C# имеется три формы этого оператора:

простой оператор присваивания: переменная = выражение;

множественное присваивание — в таком операторе последовательно справа налево нескольким переменным присваивается одно и то же значение, например: a=b=c=1;

присваивание с одновременным выполнением какой-либо операции: переменная знак_операции=выражение.

Пустой оператор — это оператор, который ничего не выполняет. Пустой оператор используется в тех случаях, когда по синтаксису языка требуется записать какой-либо оператор, а по логике программы мы не собираемся что-либо делать. Так, пустой оператор может потребоваться в операторе ветвления, когда по какой-либо ветви ничего не требуется выполнять, так же для того, чтобы определить метку для перехода в тексте программы, а иногда — для пустого тела цикла. Пустой оператор — это одиночный символ ; (точка с запятой).

Операторы перехода

Для изменения последовательного выполнения операторов используются операторы перехода. Это операторы goto, continue, break, return.

Оператор goto

Этот оператор позволяет сделать переход в пределах текущего метода. Переход возможен как по ходу выполнения программы, так и в обратном направлении. Пример:

```
goto Metka;
// Любые операторы
// .............
Metka:;
```

Здесь Metka — это идентификатор (метка), обозначающий то место в тексте программы, куда делается переход. В языке C# этот оператор используется редко. Необдуманное применение goto приводит к затруднению понимания текста программы. Как правило, если в тексте программы требуется использование этого оператора, то это

означает слабую логику в проектировании алгоритма программы.

Оператор break

Данный оператор применяется для выхода из операторов циклов (for, while, do, foreach) или оператора выбора switch. При использовании этого оператора метки не нужны, как как управление передаётся на оператор, следующий за оператором цикла или выбора.

Оператор continue

Позволяет передать управление в конец цикла. Применяется гораздо реже, чем оператор break. Как правило, всегда можно построить алгоритм решаемой задачи без использования оператора continue.

Оператор return

Обеспечивает выход из метода. Управление передаётся оператору, следующему за вызовом метода.

Оператор ветвления if

Оператор ветвления if в зависимости от условия позволяет выбрать одно из двух возможных продолжений программы. Формально в терминах языка программирования это можно записать так:

If (Условие) Оператор ветви «Да»; else Оператор ветви «Нет».

Оператор работает следующим образом: вычисляется записанное в круглых скобках выражение-условие, если оно истинно, то выполняется Оператор ветви «Да», если ложно — то Оператор ветви «Нет». После этого управление передаётся на следующий оператор.

Оператор switch

Оператор switch — это оператор для выбора одного из многих вариантов продолжения работы программы. Формальная запись этого оператора на C# выглядит так:

```
switch(Выражение)
{
case Константа1: Операторы1; break;
case Константа2: Операторы2; break;
.....
case КонстантаN: ОператорыN; break;
default:
Операторы по умолчанию; break;
}
```

Здесь Выражение — это любое выражение целого типа (чаще — просто переменная), например типа int. Так же допустимо в качестве выражения использовать выражение символьного типа char или строкового типа string.

Далее в блоке за заголовком мы видим набор конструкций вида case Константа: Операторы; break. В них Константа — это константа того же типа, что и Выражение. Операторы — любые исполняемые операторы. Количество их произвольное, оформлять блоком не требуется.

Алгоритм работы оператора switch:

1. Вычисляется значение Выражения.

2. Затем среди рассмотренных выше конструкций ищется такая, которая начинается с Константы, равной значению Выражения. Если Константа найдена, то выполняются следующие за ней Операторы. Оператор break осуществляет выход из оператора switch. Если ни одна из Констант не равна значению Выражения, то выполняется ветвь default (по умолчанию), и оператор заканчивает работу. Ветвь default в операторе switch может отсутствовать. В этом случае, если ни одна из ветвей не соответствует значению Выражения, то делается выход из оператора switch. Количество вариантов в операторе switch ни чем не ограничено. В принципе, они все могут отсутствовать (как и ветвь default), то есть вполне можно написать: switch (Выражение) {

}, но фигурные скобки за заголовком должны быть всегда.

Операторы циклов

В языке C# имеется четыре оператора цикла: for, while, do и foreach. Первые три вида цикла унаследованы из языков C/C++, а последний — foreach — является новым. Конечно, цикл можно организовать и искусственно, применяя операторы if и goto, но рекомендуется всегда использовать «настоящие» операторы цикла.

Цикл while

Слово while переводится как «пока». То есть, пока истинно некое условие, повторять цикл. Как видим — это цикл с предусловием. Вначале проверяем истинность некоторого условия, а затем, если оно истинно, выполняем операторы (один или несколько), составляющие тело цикла. Такой цикл не выполниться ни разу, если условие изначально ложно.

Формально оператор while можно записать так:

while(условие) // Заголовок

{операторы тела цикла}

Алгоритм работы оператора прост: вычисляется значение условия в заголовке оператора. Если оно истинно, то выполняются операторы тела цикла, а затем управление снова передаётся на заголовок. Если условие ложно, то оператор заканчивает работу. После этого будет выполняться оператор, следующий сразу же за оператором цикла.

Цикл do

Цикл do — это цикл с постусловием. Формальная запись:

do{

Операторы тела цикла

}while(Условие);

Цикл for

Формальная запись оператора for следующая:

for(начальные действия; условие; дополнительные действия)

{

Операторы тела цикла

}

Начальные действия — это любые операторы присваивания, записанные через запятую. Здесь, как правило, делается инициализация различных переменных, необходимых для работы в цикле for.

Условие — логическое выражение, записывается так же, как и для других операторов цикла и оператора if.

Дополнительные действия — любые записанные через запятую операторы присваивания и (или) операторы-выражения (инкремент, декремент).

Алгоритм работы оператора for:

1. Вычисляются операторы, составляющие начальные действия.

2. Вычисляется условие. Если оно ложно, то оператор заканчивает свою работу. Если условие истинно, то выполняются операторы тела цикла, потом — операторы, составляющие группу дополнительных действий, и затем снова делается проверка истинности условия. И так до тех пор, пока оно остаётся истинным. Количество операторов в начальных и в дополнительных действиях может быть любым. Любая часть в заголовке (или даже все части) могут отсутствовать: for(; ;) { операторы тела цикла }.

Цикл foreach

Это новый вид цикла, которого не было в языках C/C++. Он появился только в языке C#. Дословный перевод: «для каждого». Формальная запись этого цикла следующая:

foreach(Тип Переменная in Коллекция)

{

// тело цикла

}

где

Тип — тип данных;

Переменная — имя переменной;

in — ключевое слово;

Коллекция — имя коллекции. Коллекции рассмотрим позже. Пока можно считать, что это некий набор данных, имеющих общее имя.

Понимать оператор надо так: для каждого элемента коллекции в теле цикла надо выполнить требуемые действия. Текущий элемент коллекции, с которым выполняются действия при очередном выполнении тела цикла, хранится во временной переменной Переменная, которая должна иметь тот же тип данных, что и элементы коллекции. При работе с циклом foreach имеется целый ряд ограничений:

1. невозможно использовать без коллекций;

2. элементы коллекции доступны только по чтению, то есть их можно распечатывать, суммировать и так далее, но их нельзя изменять в цикле foreach;

3. нельзя пройти только часть коллекции, например: от начала коллекции и до её середины, или идти с шагом, отличным от 1.

В общем, цикл foreach является частным случаем цикла for.

Задание. Разработать алгоритмы решения и нарисовать блок-схемы решения следующих задач:

1. Заданы значения вещественные числа x, y, z. Вычислить значения следующих выражений:

$$a = \frac{\sqrt{|x-1|} - \sqrt[3]{|y - z^{0,25}|}}{\left(1 + \frac{x^2}{2} + \frac{y^2}{4}\right)\left(1 + z^3\right)}, \qquad b = x\left(arctg\, z + e^{-(x+y+3)}\right)$$

2. Вычислить значения y для разных x:

$$y = \begin{cases} x^3 - 2x - 3; & \text{если } x < 2 \\ x - 5; & \text{если } x = 2 \\ x^2 + x - 1; & \text{если } x > 2 \end{cases}$$

3. Вычислить:

$$\sum_{i=1}^{100} \frac{1}{i^2}$$

Практическая работа №2

Массивы. Строки в C#

Массив – это набор однотипных данных, которые располагаются в памяти последовательно друг за другом. Доступ к элементам массива осуществляется по индексу (номеру) элемента. Массивы в C# могут быть одномерными и многомерными.

Одномерные массивы

В C# объявление массива имеет следующую структуру:

тип[] имя_массива = new тип[размер массива].

Например:

int[] array = new int[5]; //массив целых чисел

string[] seasons = new string[4] {"зима","весна","лето","осень"}; //объявление массива строк и его инициализация значениями.

Оператор new можно опускать, если при объявлении массива осуществляется и его инициализация. Например:

string[] seasons = {"зима","весна","лето","осень"};

Доступ к элементам осуществляется по индексу. В C# индексация элементов массива начинается с нуля.

Многомерные массивы

Одним из случаев многомерного массива служит двумерный массив (матрица). В матрице для доступа к элементам необходимо использовать два индекса (номер строки и номер столбца).

Двумерные массивы в C# можно объявить следующим образом:

```
int[,] numbers1 = new int[2, 2]; // объявление двумерного массива
int[,,] numbers2 = new int[2, 2 ,3]; // объявление трехмерного массива
int[,] numbers3 = new int[3, 2] { {6, 0}, {5, 7}, {8, 9} }; // инициализация двумерного массива
```

Доступ к элементам двумерных массивов осуществляется с использованием двойных индексов: numbers1[1][1]=8.

Свойство массивов Length. Все массивы в языке C# являются объектами и у них есть некоторые свойства. Например свойство Length, которое возвращает количество элементов в массиве.

Пример.

```
static void Main(string[] args)
{
  int[] numbers = new int[5];
  int size = numbers.Length; // size = 5 }
```

Класс List

Специальный класс List в языке C# служит для работы со списками и представляет собой модификацию массива. Но главное его отличие от массива в том, что он динамический, т.е. его размерность можно менять в любое время, в то время как в простом массиве размер указывается при создании и сделать его больше или меньше нельзя.

Для работы с классом List в языке C# реализовано несколько методов для добавления и удаления элементов:

Add([элемент]) - добавляет элемент в конец списка.

AddRange([список элементов]) - добавляет в конец списка элементы указанного списка.

Insert([индекс],[элемент]) - вставляет элемент на позицию соответствующую индексу, все элементы «правее» будут сдвинуты на одну позицию.

InsertRange([индекс], [список элементов]) - аналогично методу Insert, но в список добавляется множество элементов.

Remove([элемент]) - удаляет первое вхождение указанного элемента из списка.

RemoveRange([индекс], [количество]) - удаляет указанное количество элементов, начиная с указанной позиции.

RemoveAt([индекс]) - удаляет элемент, который находится на указанной позиции

Clear() - удаляет все элементы списка.

Количество элементов в List возвращает свойство Count, которое аналогично свойству Length для обычного массива.

Работа со строками в C#

Для работы со строками в C# предназначен класс String, в котором разработано большое количество функций для обработки строк.

Чтобы использовать строку, ее необходимо сначала инициализировать. Рассмотрим этот процесс на простом примере:

```
static void Main(string[] args)
{ string s = "Hello, World!";
  Console.WriteLine(s); }
```

Одна из простейших операций для работы со строками - это конкатенация. Для объединения (конкатенации) строк используется оператор "+": string s = "Hello," + " World!".

Оператор "[]" используется для доступа к символу строки по индексу. Например:

string s = "Hello, World!";

char c = s[1]; // 'e'

Свойство Length возвращает длину строки.

Кроме того, при работе со строками часто используют так называемые спецсимволы или управляющие последовательности, которые мы рассмотрели ранее.

Далее рассмотрим несколько часто используемых при работе со строками методов:

Compare() - метод сравнения строк относительно алфавита. Например, строка "a" "меньше" строки "b", "bb" "больше" строки "ba". Если обе строки равны - метод возвращает "0", если первая строка меньше второй – "-1", если первая больше второй – "1". Чтобы игнорировать регистр букв, в метод нужно передать третий аргумент - "true".

ToUpper() и ToLower() - перевод строки в верхний/нижний регистр.

Contains() - поиск подстроки в строке. Данный метод принимает один аргумент – подстроку. Возвращает True, если строка содержит подстроку, в противном случае – False.

IndexOf() - возвращает индекс первого символа подстроки, которую содержит строка. Данный метод принимает один аргумент – подстроку. Если строка не содержит подстроки, метод возвращает "-1".

StartsWith() и EndsWith() - дают ответ на вопрос "начинается/заканчивается ли строка указанной подстрокой?".

Insert() - позволяет вставить подстроку в строку, начиная с указанной позиции. Принимает два аргумента – позиция и подстрока.

Remove() - обрезает строку начиная с указанной позиции. Метод Remove() принимает один аргумент – позиция, начиная с которой обрезается строка. В метод Remove() можно передать и второй аргумент – количество обрезаемых символов. Remove(3,5) – удалит из строки пять символов начиная с 3-го.

Substring() - позволяет получить подстроку из строки, начиная с указанной позиции. Метод принимает один аргумент – позиция, с которой будет начинаться новая подстрока. В метод Substring() можно передать и второй аргумент – длина подстроки. Substring (3, 5) – возвратит подстроку длиной в 5 символов начиная с 3-й позиции строки.

Replace() - заменяет в строке все подстроки указанной новой подстрокой. Метод принимает два аргумента – подстрока, которую нужно заменить и новая подстрока, на которую будет заменена первая.

ToCharArray() - преобразует строку в массив символов. Метод возвращает массив символов указанной строки.

Split() - разбивает строку по указанному символу на массив подстрок. Принимает один аргумент - символ, по которому будет разбита строка. Возвращает массив строк.

Задание. Разработать алгоритмы решения следующих задач:

1. Дан одномерный массив, состоящий из N целочисленных элементов.
   Ввести массив с клавиатуры.
   Найти максимальный элемент.
   Вычислить среднеарифметическое элементов массива.
   Вывести массив на экран в обратном порядке.
2. Дан двумерный массив размерностью 4×6, заполненный целыми числами с клавиатуры. Сформировать одномерный массив, каждый элемент которого равен количеству элементов соответствующей строки, больших данного числа.
3. Дана строка, содержащая английский текст. Найти количество слов, начинающихся с буквы b.

Практическая работа №3

Функции. Файлы

Разработка пользовательских функций в C#

В программировании часто встречаются случаи, когда в разных частях программы встречается одна и та же последовательность операторов, решающая одинаковую задачу. Для большей компактности и наглядности, языки программирования позволяют выделять из текста программы повторяющийся фрагмент и записывать его только один раз, представив самостоятельным программным объектом, имеющим собственное имя, при этом допускается многократное обращение к этому объекту в разных местах программы. Объект такого рода называется подпрограммой. В языке C# подпрограммами являются функции, которые также называют методами. Любая функция в C# может быть объявлена только в рамках класса, так как C# - полностью объектно-ориентированный язык программирования. Объявление функции имеет следующую структуру:

[модификатор доступа] [тип возвращаемого значения] [имя функции] ([аргументы])

{// тело функции}

Модификатор доступа (области видимости) может быть public, private, protected, internal.

Между модификатором и типом может стоять ключевое слово static, что означает, что функция будет статичной. Из статичной функции можно вызывать другие функции, если они тоже статичные.

Функция может возвращать значение или не возвращать. Если функция, например, возвращает целое число, то нужно указать тип int. Если функция не возвращает никакого значения, то для этого используется ключевое слово void.

Аргументы – это те данные, которые необходимы для выполнения функции. Аргументы записываются в формате [тип] [идентификатор]. Если аргументов несколько, они отделяются запятой. Аргументы могут отсутствовать.

Первая строка функции, где указываются тип, имя, аргументы и т.д. называется заголовком функции.

Оператор return

Когда встречается этот оператор, происходит выход из функции и код ниже (если он есть) выполняться не будет (например, в функцию передан такой аргумент, при котором нет смысла выполнять функцию). Он похож на оператор break, который используется для выхода из циклов. Этот оператор также можно использовать и в функциях, которые не возвращают значение. Оператор return допустимо использовать несколько раз в функции, но этого делать не рекомендуется.

Работа с файлами в Си-шарп. Классы StreamReader и StreamWriter

Файл – это набор данных, который хранится на внешнем запоминающем устройстве (например, на жестком диске). Файл имеет имя и расширение. Расширение позволяет идентифицировать, какие данные и в каком формате хранятся в файле. Под работой с файлами подразумевается: создание файлов, удаление файлов, чтение данных, запись данных, изменение параметров файла (имя, расширение…) и др.

В C# есть пространство имен System.IO, в котором реализованы все необходимые классы для работы с файлами. Чтобы подключить это пространство имен, необходимо в самом начале программы добавить строку using System.IO. Для использования кодировок еще можно добавить пространство using System.Text.

Теперь рассмотрим процесс создания файла. Для создания пустого файла, в классе File есть метод Create(). Он принимает один аргумент – путь. Ниже приведен пример создания пустого текстового файла new_file.txt на диске D:

```
static void Main(string[] args)
{   File.Create("D:\\new_file.txt");}
```

Если файл с таким именем уже существует, он будет переписан на новый пустой файл.

Метод WriteAllText() создает новый файл (если такого нет), либо открывает существующий и записывает текст, заменяя всё, что было в файле:

```
static void Main(string[] args)
{   File. WriteAllText("D:\\new_file.txt", "текст");}
```

Метод AppendAllText() работает, как и метод WriteAllText() за исключением того, что новый текст дописывается в конец файла, а не перезаписывает файл:

```
static void Main(string[] args)
{       File.AppendAllText("D:\\new_file.txt", "текст метода AppendAllText ()"); //допишет текст в конец файла}
```

Для удаления файла используется метод Delete(), который удаляет файл по указанному пути:

```
static void Main(string[] args)
{   File.Delete("d:\\test.txt"); //удаление файла}
```

Кроме того, чтобы читать/записывать данные в файл в C# можно использовать потоки. Поток – это абстрактное представление данных (в байтах), которое облегчает работу с ними. В качестве источника данных может быть файл, устройство ввода-вывода, принтер.

Класс Stream является базовым классом для всех потоковых классов в C#. Для работы с файлами нам понадобится класс FileStream (файловый поток).

FileStream - представляет поток, который позволяет выполнять операции чтения/записи в файл.

```
static void Main(string[] args)
{   FileStream file = new FileStream("d:\\test.txt", FileMode.Open, FileAccess.Read); //открывает файл только на чтение}
```

Режимы открытия FileMode:

Append – открывает файл (если существует) и переводит указатель в конец файла (данные будут дописываться в конец), или создает новый файл. Данный режим возможен только при режиме доступа FileAccess.Write.

Create - создает новый файл (если существует – заменяет).

CreateNew – создает новый файл (если существует – генерируется исключение).

Open - открывает файл (если не существует – генерируется исключение).

OpenOrCreate – открывает файл, либо создает новый, если его не существует.

Truncate – открывает файл, но все данные внутри файла затирает (если файла не существует – генерируется исключение)

Режим доступа FileAccess:

Read – открытие файла только на чтение. При попытке записи генерируется исключение.

Write - открытие файла только на запись. При попытке чтения генерируется исключение.

ReadWrite - открытие файла на чтение и запись.

Чтение из файла. Для чтения данных из потока нам понадобится класс StreamReader. В нем реализовано множество методов для удобного считывания данных. Ниже приведена программа, которая выводит содержимое файла на экран:

```
static void Main(string[] args)
{
  FileStream file1 = new FileStream("d:\\test.txt", FileMode.Open); //создаем файловый поток
  StreamReader reader = new StreamReader(file1); // создаем «потоковый читатель» и связываем его с файловым потоком
  Console.WriteLine(reader.ReadToEnd()); //считываем все данные с потока и выводим на экран
  reader.Close(); //закрываем поток
```

```
  Console.ReadLine();
}
```

Метод ReadToEnd() считывает все данные из файла. ReadLine() – считывает одну строку (указатель потока при этом переходит на новую строку, и при следующем вызове метода будет считана следующая строка).

Свойство EndOfStream указывает, находится ли текущая позиция в потоке в конце потока (достигнут ли конец файла). Возвращает true или false.

Запись в файл. Для записи данных в поток используется класс StreamWriter. Пример записи в файл:

```
static void Main(string[] args)
{     FileStream file1 = new FileStream("d:\\test.txt", FileMode.Create); //создаем файловый поток
   StreamWriter writer = new StreamWriter(file1); //создаем «потоковый писатель» и связываем его с файловым потоком
    writer.Write("текст"); //записываем в файл
    writer.Close(); //закрываем поток. Не закрыв поток, в файл ничего не запишется }
```

Метод WriteLine() записывает в файл построчно (то же самое, что и простая запись с помощью Write(), только в конце добавляется новая строка).

Нужно всегда помнить, что после работы с потоком, его нужно закрыть (освободить ресурсы), использовав метод Close().

Кодировка, в которой будут считываться/записываться данные указывается при создании StreamReader/StreamWriter:

```
static void Main(string[] args)
{
  FileStream file1 = new FileStream("d:\\test.txt", FileMode.Open);
  StreamReader reader = new StreamReader(file1, Encoding.Unicode);
  StreamWriter writer = new StreamWriter(file1, Encoding.UTF8);
}
```

Кроме того, при использовании StreamReader и StreamWriter можно не создавать отдельно файловый поток FileStream, а сделать это сразу при создании StreamReader/StreamWriter:

```
static void Main(string[] args)
{ StreamWriter writer = new StreamWriter("d:\\test.txt"); //указываем путь к файлу
  writer.WriteLine("текст");
  writer.Close(); }
```

Задание. Разработать алгоритмы решения задач:

1. Даны действительные числа s,t. Получить g(1.2,s) + g(t,s) - g(2s-1,st), где $g(a,b)=\frac{a^2-b^2}{2\cdot a\cdot b-a-b}+(a+b)\cdot\sqrt{\frac{|a+b|}{2}}$. Решение осуществить путем создания подпрограмм-функций.
2. Вычислить с использованием функции наименьшие элементы в строке и сумму номеров строк и столбцов, в которых они расположены, для матрицы A(10,15). Результаты формировать в одномерных массивах M(10) и S(10). Ввод и вывод массивов выполнить в отдельных функциях.
3. Дан файл f, элементы которого являются действительными числами. Найти модуль суммы и квадрат произведения элементов файла f.

Практическая работа №4

Классы. Указатель this

Классы в C#

Класс — это конструкция, которая позволяет создавать собственные настраиваемые типы путем группирования переменных других типов, методов и событий. В языке C# объявление класса начинается с ключевого слова class. Общая структура объявления выглядит следующим образом:

[модификатор доступа] class [имя_класса]

{ //тело класса }

В языке C# есть несколько модификаторов доступа для классов:

public - доступ к типу или члену возможен из любого другого кода в той же сборке или другой сборке, ссылающейся на него;

private - доступ к типу или члену возможен только из кода в том же классе или структуре;

protected - доступ к типу или члену возможен только из кода в том же классе или структуре либо в классе, производном от этого класса;

internal - доступ к типу или члену возможен из любого кода в той же сборке, но не из другой сборки.

Сборка (assembly) – это готовый функциональный модуль в виде exe либо dll файла (файлов), который содержит скомпилированный код для .NET. Сборка предоставляет возможность повторного использования кода. При объявлении класса модификатор доступа можно не указывать, при этом будет применяться режим по умолчанию internal.

Класс следует объявлять внутри пространства имен namespace, но за пределами другого класса (возможно также объявление класса внутри другого).

Члены класса

В классах есть члены, представляющие их данные и поведение. Члены класса включают все члены, объявленные в этом классе, а также все члены (кроме конструкторов и методов завершения), объявленные во всех классах в иерархии наследования данного класса. Закрытые члены в базовых классах наследуются, но недоступны из производных классов.

Далее перечислены виды членов, которые могут содержатся в классе:

Поля - Поля являются переменными, объявленными в области класса. Поле может иметь встроенный числовой тип или быть экземпляром другого класса. Например, в классе календаря может быть поле, содержащее текущую дату.

Константы - Константы — это поля или свойства, значения которых устанавливаются во время компиляции и не изменяются.

Свойства - Свойства — это методы класса. Доступ к ним осуществляется так же, как если бы они были полями этого класса. Свойство может защитить поле класса от изменений (независимо от объекта).

Методы - Методы определяют действия, которые может выполнить класс. Методы могут принимать параметры, предоставляющие входные данные, и возвращать выходные данные посредством параметров. Методы могут также возвращать значения напрямую, без использования параметров.

События - События предоставляют другим объектам уведомления о различных случаях, таких как нажатие кнопки или успешное выполнение метода. События определяются и переключаются с помощью делегатов.

Операторы - Перегруженные операторы считаются членами класса. При перегрузке оператора его следует определять как открытый статический метод в классе. Предопределенные операторы (+, *, < и т. д.) не считаются членами.

Индексаторы - Индексаторы позволяют индексировать объекты аналогично массивам.

Конструкторы - Конструкторы — это методы, которые вызываются при создании объекта. Зачастую они используются для инициализации данных объекта.

Методы завершения - Методы завершения очень редко используются в C#. Они являются методами, вызываемыми средой выполнения, когда объект нужно удалить из

памяти. Они обычно применяются для правильной обработки ресурсов, которые должны быть высвобождены.

Вложенные типы - Вложенными типами являются типы, объявленные в другом типе. Вложенные типы часто применяются для описания объектов, использующихся только типами, в которых эти объекты находятся.

Создание объектов

Рассмотрим создание объектов класса на простом примере. Создание объекта осуществляется с помощью ключевого слова new и имени класса:

```
namespace HelloWorld
{
  class Student
  {
    private string firstName;
    private string lastName;
    private int age;
    public string group;}
  class Program
  {
    static void Main(string[] args)
    {
      Student student1 = new Student(); //создание объекта student1 класса Student
      Student student2 = new Student();     }    }  }
```

Доступ к членам объекта осуществляется при помощи оператора точка «.» :

```
static void Main(string[] args)
{
  Student student1 = new Student();
  Student student2 = new Student();
  student1.group = "Group1";
  student2.group = "Group2";
  Console.WriteLine(student1.group); // выводит на экран "Group1"
  Console.Write(student2.group);
  Console.ReadKey(); }
```

Такие поля класса Student, как firstName, lastName и age указаны с модификатором доступа private, поэтому доступ к ним будет запрещен вне класса:

```
static void Main(string[] args)
{Student student1 = new Student();
  student1.firstName= "Nikolay"; //ошибка, нет доступа к полю firstName. Программа
```

не скомпилируется }

Константы

Константы-члены класса ничем не отличаются от простых констант внутри методов. Константа объявляется с помощью ключевого слова const. Пример объявления константы:

```
class Math
{   private const double Pi = 3.14;}
```

Методы

Методы являются основными членами класса. Выделяют простые методы и статистические.

Статический метод – это метод, который не имеет доступа к полям объекта, и для вызова такого метода не нужно создавать экземпляр (объект) класса, в котором он объявлен.

Простой метод – это метод, который имеет доступ к данным объекта, и его вызов выполняется через объект. Простые методы служат для обработки внутренних данных объекта.

Приведем пример использования простого метода. Класс Телевизор, у него есть поле switchedOn, которое отображает состояние включен/выключен, и два метода – включение и выключение:

```
class TVSet
{ private bool switchedOn;
  public void SwitchOn()
  { switchedOn = true;   }
  public void SwitchOff()
  {     switchedOn = false;   } }
class Program
{   static void Main(string[] args)
  {     TVSet myTV = new TVSet();
    myTV.SwitchOn(); // включаем телевизор, switchedOn = true;
    myTV.SwitchOff(); // выключаем телевизор, switchedOn = false;   }}
```

Чтобы вызвать простой метод, перед его именем, указывается имя объекта. Для вызова статического метода необходимо указывать имя класса. Статические методы, обычно, выполняют какую-нибудь глобальную, общую функцию, обрабатывают «внешние данные». Например, сортировка массива, обработка строки, возведение числа в степень и другое. Статический метод не имеет доступа к нестатическим полям класса.

Конструкторы. Указатель this

Конструктор – это метод класса, предназначенный для инициализации объекта при его создании.

Особенностью конструктора, как метода, является то, что его имя всегда совпадает с именем класса, в котором он объявляется. При этом, при объявлении конструктора, не нужно указывать возвращаемый тип, даже ключевое слово void. Конструктор следует объявлять как public, иначе объект нельзя будет создать. В классе, в котором не объявлен ни один конструктор, существует неявный конструктор по умолчанию, который вызывается при создании объекта с помощью оператора new.

Объявление конструктора имеет следующую структуру:

```
public [имя_класса] ([аргументы])
{  // тело конструктора }
```

Ключевое слово this

Указатель this - это указатель на объект, для которого был вызван нестатический метод. Ключевое слово this обеспечивает доступ к текущему экземпляру класса. Классический пример использования this - это его использование в конструкторах, при одинаковых именах полей класса и аргументов конструктора. Ключевое слово this это что-то вроде имени объекта, через которое мы имеем доступ к текущему объекту.

Несколько конструкторов

В классе возможно указывать множество конструкторов, главное чтобы они отличались сигнатурами. Сигнатура, в случае конструкторов, - это набор аргументов. Например, нельзя создать два конструктора, которые принимают два аргумента типа int.

Пример использования нескольких конструкторов:

```
class Car
{
  private double mileage;
  private double fuel;
  public Car()
  {
    mileage = 0;
```

```
    fuel = 0;   }
  public Car(double mileage, double fuel)
  {
    this.mileage = mileage;
    this.fuel = fuel;
  }
}
class Program
{
  static void Main(string[] args)
  {
  Car newCar = new Car(); // создаем автомобиль с параметрами по умолчанию, 0 и 0
    Car newCar2 = new Car(100, 50); // создаем автомобиль с указанными
//параметрами   } }
```

Если в классе определен один или несколько конструкторов с параметрами, мы не сможем создать объект через неявный конструктор по умолчанию:

```
class Car
{
  private double mileage;
  private double fuel;
  public Car(double mileage, double fuel)
  {
    this.mileage = mileage;
    this.fuel = fuel;
  }
}
class Program
{
  static void Main(string[] args)
  {
    Car newCar = new Car(100, 50);
    Car newCar2 = new Car(); // ошибка, в классе не определен конструктор без
параметров
  } }
```

Свойства

Свойство — это особый вид методов, обеспечивающих доступ к полям класса. Для каждого свойства может быть два таких метода: один — для чтения значения поля (get), другой — для записи в него какого-то значения (set). В свойстве могут быть определены оба метода или только какой-либо один. Как правило, одно свойство используют для доступа только к одному полю. Ни для каких других задач свойства не рекомендуется применять.

Формальная запись объявления свойства имеет следующий вид:

```
Доступ Тип Имя_свойства
{
get
{
return Результат
}
set
{
Поле_класса = value;
}
```

```
}
```

Использование свойства в программе:

Имя_объекта_класса.Имя_свойства = Выражение;

или

Переменная = Имя_объекта_класса.Имя_свойства;

где Доступ — обычно это слово public (общий доступ), но допустимо использовать protected, private, internal, а также static и new;

Тип — тип свойства. Должен совпадать с типом поля, с которым связано свойство;

Имя_свойства — задаётся по общим правилам для идентификаторов;

get — ключевое слово, оно определяет метод для получения (чтения) значения;

set — ключевое слово, оно определяет метод для изменения (записи) значения;

value — ключевое слово, используется в методе set как неявный параметр и содержит значение, которое необходимо присвоить полю;

Результат — обычно это значение поля класса (у нас далее обозначается как Поле_класса), связанного со свойством;

Переменная и Выражение — должны иметь тот же тип, что и тип свойства.

Пример. Для класса, моделирующего работу с окружностью, создадим свойство для изменения радиуса окружности.

```
using System;
namespace Prim_Svojstvo
{
public class Okr
{
int x, y, r; // координаты центра окружности и
её радиус
public Okr()
{
x = y = 100;
r = 10;
}
public Okr(int x0, int y0, int r0)
{
x = x0;
y = y0;
r = r0;
}
public void print()
{
Console.WriteLine("x = {0} y = {1} r = {2}",
x, y, r);
}
public int Radius
{
set
{
if(value > 0)
r = value;
else
{
Console.WriteLine("Недопустимое значение для поля: {0}", value);
Console.WriteLine("Поле сохраняет прежнее значение: {0}", r);
} }
```

```
get
{
return r;
}  }
}
class Program
{
public static void Main(string[] args)
{
Okr x = new Okr(25, 45, 5);
x.print();
x.Radius = 4;
x.print();
int r2 = x.Radius * 2;
Console.WriteLine("Удвоенное значение радиуса
r2 = {0}", r2);
Console.Write("Press any key to continue . .
. ");
Console.ReadKey(true);
} }
}
```

Свойства применяют для доступа к полям, которые, как правило, всегда делают закрытыми (private) или защищёнными (protected).

Задание. Разработать алгоритм решения следующей задачи:

Разработать класс: **Abiturient**: Фамилия, Имя, Отчество, Адрес, Оценки. Создать массив объектов. Вывести:

а) список абитуриентов, имеющих неудовлетворительные оценки;

б) список абитуриентов, сумма баллов у которых не меньше заданной;

в) выбрать N абитуриентов, имеющих самую высокую сумму баллов, и список абитуриентов, имеющих полупроходной балл.

Практическая работа №5

Перегрузка операций и методов

Перегрузка операций

В языке C# как и в C++ можно при желании перегружать операции для работы с объектами своих классов. Перегрузка операций, как и перегрузка методов, является одной из форм полиморфизма.

При перегрузке операций в C# существует ряд ограничений:

нельзя использовать свои знаки операций;

нельзя изменить приоритет операции, например, если для чисел приоритет операции умножения (*) выше, чем сложения (+), то и в классе пользователя при перегрузке этих операций сохраняется то же старшинство действий;

метод, реализующий перегрузку какой-либо операции, должен быть статическим и открытым;

параметры можно передавать в метод для реализации операции только по значению;

нельзя перегружать ни какие формы операции присваивания (=, += и т.д.);

операции сравнения необходимо реализовывать парами (симметричными по смыслу): <= и >= < и > == и !=.

Теперь рассмотрим ряд примеров по перегрузке операций. Для большей конкретности создадим какой-нибудь класс, например, класс, моделирующий обыкновенные дроби. Вот как может выглядеть начало этого класса:

```
public class Drobi
{
int a; // Числитель
int b; // Знаменатель
// Конструкторы и другие методы
.....
}
```

Перегрузка унарных операций

В языке C# можно перегрузить следующие унарные операции:

+ (унарный плюс)
- (унарный минус)
!
~
++
--
true
false

Формально перегрузка операции записывается таким образом:

```
public static Тип_результата operator Знак_операции
(Формальный_Параметр)
{// тело метода, реализующего перегрузку операции}
```

Как видим, метод, перегружающий операцию, всегда объявляется открытым (public) и статичным (static). Признаком того, что делается именно перегрузка операции, служит ключевое слово operator, за которым должен располагаться знак перегружаемой операции. В качестве формального параметра при перегрузке унарной операции может выступать только объект ого же типа (класса), что и класс, для которого мы делаем перегрузку операции. Формальный параметр передаётся только по значению.

Приведём пример перегрузки унарной операции для класса Drobi.

Пример. Перегрузка операции - (смена знака):

```
public static Drobi operator - (Drobi x)
{
Drobi t = new Drobi();
t.a = - x.a;
t.b = x.b;
return t;
}
```

Здесь в методе создаётся новый объект t типа Drobi и затем вычисляются значения полей этого объекта. Метод возвращает результат — переменную t типа Drobi, исходный объект остаётся неизменным. Пример использования операции:

```
Drobi x = new Drobi(2,5);
Drobi y = new Drobi();
y = -x;
```

После этих действий переменная y будет иметь значение -2/5.

Стоит отметить, что операции инкремент и декремент имеют две формы: префиксную (например, ++i) и постфиксную (например, i++). Но при перегрузке этих операций (инкремент и декремент) обе формы вызывают один и тот же метод. Поэтому перегружаем метод один раз, а затем используем в любой из форм.

Перегрузка бинарных операций

В языке C# можно перегрузить следующие двухместные операции: +, -, *, /, %, &, |, ^, <<, >>, ==, !=, <, >, >=, <=.

Формально перегрузка бинарных операций записывается следующим образом:

```
public static Тип_результата operator Знак_операции
```

(Формальный_Параметр1, Формальный_Параметр2)

{// тело метода, реализующего перегрузку операции}

Любой метод, реализующий перегрузку бинарной операции, также является открытым и статичным. При этом хотя бы один из двух формальных параметров должен иметь тип класса, для которого делается эта перегрузка.

Перегрузка методов в C#

Теперь рассмотрим еще один способ реализации полиморфизма в C# – перегрузка методов.

Перегрузка методов – это объявление в классе методов с одинаковыми именами, но с различными параметрами.

Пример того, как может быть перегружен метод:

```
public void SomeMethod()
{   // тело метода}
public void SomeMethod(int a) // от первого отличается наличием параметра
{   // тело метода}
public void SomeMethod(string s) // от второго отличается типом параметра
{   // тело метода}
public int SomeMethod(int a, int b) // от предыдущих отличается количеством параметров (плюс изменен тип возврата)
{   // тело метода
  return 0;}
```

Один из наиболее часто встречающихся примеров перегрузки методов - использование нескольких конструкторов, например, без параметров и с параметрами.

Задание. Разработать алгоритм решения следующих задач:

1. Для строки символов реализовать перегрузку операций:
   сравнение строк (операция ==);
   удаление из строки заданного символа (операция –).

   Кроме того, членом класса сделать функцию с именем strset() для удаления из первой строки всех символов, встречающихся во второй строке.
2. Определить класс-строку. В класс включить два конструктора: для определения класса строки строкой символов и путем копирования другой строки (объекта класса строки). Определить операции над строками:
   >> перевертывание строки (запись символов в обратном порядке);
   ++ нахождение наименьшего слова в строке.

Практическая работа №6

Наследование

Теперь рассмотрим один из базовых принципов объектно-ориентированного программирования – наследование, которое позволяет создавать новый класс на базе другого. Класс, на базе которого создается новый класс, называется базовым, а базирующийся новый класс – наследником или производным классом. В класс-наследник из базового класса переходят поля, свойства, методы и другие члены класса.

Объявление нового класса, который будет наследовать другой класс, выглядит так:

```
class [имя_класса] : [имя_базового_класса]
{   // тело класса }
```

Приведем простой пример использования наследования. На основе базового класса Животное создаются два класса Собака и Кошка, в эти два класса переходит свойство Имя животного:

```
class Animal
{   public string Name { get; set; }}
class Dog : Animal
{   public void Guard()
```

```
  {    // собака охраняет   }}
class Cat : Animal
{  public void CatchMouse()
  {    // кошка ловит мышь   } }
class Program
{  static void Main(string[] args)
  {    Dog dog1 = new Dog();
    dog1.Name = "Барбос"; // называем пса
    Cat cat1 = new Cat();
    cat1.Name = "Барсик"; // называем кота
    dog1.Guard(); // отправляем пса охранять
    cat1.CatchMouse(); // отправляем кота на охоту    } }
```

Вызов конструктора базового класса

В базовом классе и классе-наследнике могут быть объявлены конструкторы. Конструктор базового класса будет создавать ту часть объекта, которая принадлежит базовому классу (ведь из базового класса о наследнике ничего неизвестно), а конструктор из наследника будет создавать свою часть.

Когда конструктор определен только в наследнике, то здесь всё просто – при создании объекта сначала вызывается конструктор по умолчанию базового класса, а затем конструктор наследника.

Когда конструкторы объявлены и в базовом классе, и в наследнике – нам необходимо вызывать их оба. Для вызова конструктора базового класса используется ключевое слово base. Объявление конструктора класса-наследника с вызовом базового конструктора имеет следующую структуру:

[имя_конструктора_класса-наследника] ([аргументы]) : base ([аргументы])
{ // тело конструктора }

В базовый конструктор передаются все необходимые аргументы для создания базовой части объекта. Рассмотрим это на примере. Пусть имеется тот же класс Животное и класс Попугай. В классе Животное есть только свойство Имя, и конструктор, который позволяет установить это имя. В классе Попугай есть свойство Длина клюва и конструктор, в котором мы задаем эту длину. При создании объекта Попугай мы указываем два аргумента – имя и клюв, и дальше аргумент Имя передается в базовый конструктор, он вызывается, и после его работы выполнение передается конструктору класса Попугай, где устанавливается длина:

```
class Animal
{
  public string Name { get; set; }
  public Animal(string name)
  {
    Name = name;
  }
}
class Parrot : Animal
{
  public double BeakLength { get; set; } // длина клюва

  public Parrot(string name, double beak) : base(name)
  {
    BeakLength = beak;
  }
}
class Dog : Animal
```

```
{
  public Dog(string name) : base (name)
   {
     // здесь может быть логика создания объекта Собака
   }
}
class Program
{
  static void Main(string[] args)
  {
    Parrot parrot1 = new Parrot("Кеша", 4.2);
    Dog dog1 = new Dog("Барбос");
  }
}
```

Доступ к членам базового класса из класса-наследника

Здесь стоит отметить, что в классе-наследнике мы можем получить доступ к членам базового класса, которые объявлены как public, protected, internal. Члены базового класса с модификатором доступа private также переходят в класс-наследник, но к ним могут иметь доступ только члены базового класса. Например, свойство, объявленное в базовом классе, которое управляет доступом к закрытому полю, будет работать корректно в классе-наследнике, но отдельно получить доступ к этому полю из класса-наследника мы не сможем.

Задание. Разработать алгоритм решения следующей задачи:

Требуется создать базовый класс «Транспортное средство» и определить общие и специфические методы для данного класса. Создать производные классы «Автомобиль», «Велосипед», «Повозка», в которые добавить свойства и методы. Часть методов переопределить. Создать массив объектов базового класса и заполнить объектами производных классов. Предусмотреть передачу аргументов конструкторам базового класса. Подсчитать время и стоимость перевозки пассажиров и грузов каждым транспортным средством.

Практическая работа №7

Виртуальные методы. Абстрактные классы

Виртуальный метод – это метод, который может быть переопределен в классе наследнике.

Виртуальный метод объявляется при помощи ключевого слова virtual:

[модификатор доступа] virtual [тип] [имя метода] ([аргументы])

{ // тело метода }

Объявив виртуальный метод, мы теперь можем переопределить его в классе наследнике. Для этого используется ключевое слово override:

[модификатор доступа] override [тип] [имя метода] ([аргументы])

{ // новое тело метода}

Вызов базового метода

Бывает так, что функционал метода, который переопределяется, в базовом классе мало отличается от функционала, который должен быть определен в классе - наследнике. В таком случае, при переопределении, мы можем вызвать сначала этот метод из базового класса, а дальше дописать необходимый функционал. Это делается при помощи ключевого слова base.

Абстрактные классы

Абстрактный класс – это класс объявленный с ключевым словом abstract:

abstract class [имя_класса] { //тело}

Такой класс имеет следующие особенности:

нельзя создавать экземпляры (объекты) абстрактного класса;

абстрактный класс может содержать как абстрактные методы/свойства, так и обычные;

в классе наследнике должны быть реализованы все абстрактные методы и свойства, объявленные в базовом классе.

В самом по себе абстрактном классе, от которого никто не наследуется, смысла нет, так как нельзя создавать его экземпляры. В абстрактном классе обычно реализуется некоторая общая часть нескольких сущностей или другими словами - абстрактная сущность, которая, как объект, не может существовать, и эта часть необходима в классах наследниках.

Абстрактные методы

Абстрактный метод – это метод, который не имеет своей реализации в базовом классе, и он должен быть реализован в классе-наследнике. Абстрактный метод может быть объявлен только в абстрактном классе.

Разница между виртуальным и абстрактным методами заключается в следующем:

виртуальный метод может иметь свою реализацию в базовом классе, абстрактный – нет (тело пустое);

абстрактный метод должен быть реализован в классе наследнике, виртуальный метод переопределять необязательно.

Объявление абстрактного метода происходит при помощи ключевого слова abstract, и при этом фигурные скобки опускаются, точка с запятой ставится после заголовка метода:

[модификатор доступа] abstract [тип] [имя метода] ([аргументы]);

Реализация абстрактного метода в классе наследнике происходит так же, как и переопределение метода – при помощи ключевого слова override:

[модификатор доступа] override [тип] [имя метода] ([аргументы])

{ // реализация метода}

Абстрактные свойства

Создание абстрактных свойств не сильно отличается от методов:

protected [тип] [поле, которым управляет свойство];

[модификатор доступа] abstract [тип] [имя свойства] { get; set; }

Реализация в классе-наследнике:

[модификатор доступа] override [тип] [имя свойства]

{ get { тело аксессора get }

set { тело аксессора set }}

Задание. Разработать алгоритм решения следующих задач:

1. Разработать программу с использованием наследования классов, реализующую классы: графический объект, круг, квадрат. Используя виртуальные функции, выведите на экран размер и координаты графического объекта.
2. Создать абстрактный базовый класс уравнения с виртуальной функцией печать корней уравнения. Создать производные классы: линейное уравнение и квадратное уравнение. Определить функцию вычисления корней уравнений. Для проверки определить массив указателей на абстрактный класс, которым присваиваются адреса различных объектов.

Практическая работа №8

Обработка исключительных ситуаций

Обработка исключений. Оператор try-catch

Обработка исключений – это описание реакции программы на исключения во время выполнения программы. Реакцией программы может быть корректное завершение работы программы, вывод информации об ошибке и запрос повторения действия (при вводе данных).

Примерами исключений может быть:

деление на ноль;

конвертация некорректных данных из одного типа в другой;

попытка открыть файл, которого не существует;

доступ к элементу вне рамок массива;

исчерпывание памяти программы и др.;

Для обработки исключений в C# используется оператор try-catch. Он имеет следующую структуру:

```
try
{  //блок кода, в котором возможно исключение}
catch ([тип исключения] [имя])
{  //блок кода – обработка исключения}
```

Работает это следующим образом. Выполняется код в блоке try, и, если в нем происходит исключение типа, соответствующего типу, указанному в catch, то управление передается блоку catch. При этом, весь оставшийся код от момента выбрасывания исключения до конца блока try не будет выполнен. После выполнения блока catch, оператор try-catch завершает работу.

Указывать имя исключения не обязательно. Исключение представляет собою объект, и к нему мы имеем доступ через это имя. С этого объекта мы можем получить, например, стандартное сообщение об ошибке (Message), или трассировку стека (StackTrace), которая поможет узнать место возникновения ошибки. В этом объекте хранится детальная информации об исключении.

Если тип выброшенного исключения не будет соответствовать типу, указанному в catch – исключение не обработается, и программа завершит работу аварийно.

Ниже приведен пример программы, в которой используется обработка исключения

Типы исключений

Ниже приведены некоторые из часто встречаемых типов исключений:

Exception – базовый тип всех исключений. Блок catch, в котором указан тип Exception будет «ловить» все исключения.

FormatException – некорректный формат операнда или аргумента (при передаче в метод).

NullReferenceException - в экземпляре объекта не задана ссылка на объект, объект не создан.

IndexOutOfRangeException – индекс вне рамок коллекции.

FileNotFoundException – файл не найден.

DivideByZeroException – деление на ноль

Несколько блоков catch

Одному блоку try может соответствовать несколько блоков catch:

```
try
{  //блок1}
catch (FormatException)
{  //блок-обработка исключения 1}
catch (FileNotFoundException)
{  //блок-обработка исключения 2}
```

В зависимости от того или другого типа исключения в блоке try, выполнение будет передано соответствующему блоку catch.

Блок finally

Оператор try-catch также может содержать блок finally. Особенность блока finally в том, что код внутри этого блока выполнится в любом случае, в независимости от того, было ли исключение или нет.

```
try
{  //блок1}
```

```
catch (Exception)
{  //обработка исключения}
finally
{  //блок кода, который выполнится обязательно}
```

Выполнение кода программы в блоке finally происходит в последнюю очередь. Сначала try, затем finally или catch-finally (если было исключение). Обычно он используется для освобождения ресурсов. Классическим примером использования блока finally является закрытие файла.

Блок finally гарантирует выполнение кода, несмотря ни на что. Даже если в блоках try или catch будет происходить выход из метода с помощью оператора return – finally выполнится. Операторы try-catch также могут быть вложенными. Внутри блока try либо catch может быть еще один try-catch. Обработка исключений, в первую очередь, нам понадобится при работе с файлами.

Задание. Разработать алгоритм решения следующих задач:

1. Создать три массива a, b и c размерами соответственно n1, n2 и n3 (n1≠n2≠n3). В массив a занести значения функции $f(x)=\ln(x-1), x\in[0;10], \Delta x=0{,}5$ (при возникновении исключения заносить нули). Массив b заполнить случайными числами (среди них должны быть и отрицательные числа и нули). Массив c формируется следующим образом: $c_i=a_i+1/b_i$. Предусмотреть и обработать возникающие при этом исключительные ситуации (деление на ноль, корень из отрицательного числа, арифметическое переполнение, выход за пределы диапазона индексов массива и т.п.).
2. Проверить правильность перевода из двоичной системы счисления в троичную. Написать функцию, которая должна генерировать исключение некорректных значений параметров. Приведите пример некорректного использования.

Раздел 2. Лабораторные работы

Весь необходимый для выполнения лабораторных работ теоретический материал приведен в Разделе 1 и в источниках информации из списка литературы.

В рамках выполнения лабораторных работ необходимо решить поставленные в лабораторных работах задачи и подготовить отчет.

Лабораторная работа №1

Введение в объектно-ориентированное программирование

Задание. Написать программу на C# для решения следующих задач:

1. Заданы значения вещественные числа x, y, z. Вычислить значения следующих выражений:

$$a=\frac{\sqrt{|x-1|}-\sqrt[3]{|y-z^{0,25}|}}{\left(1+\frac{x^2}{2}+\frac{y^2}{4}\right)\left(1+z^3\right)},\qquad b=x\left(arctg\, z+e^{-(x+y+3)}\right)$$

2. Вычислить значения y для разных x:

$$y=\begin{cases} x^3-2x-3; & если\quad x<2\\ x-5; & если\quad x=2\\ x^2+x-1; & если\quad x>2\end{cases}$$

3. Вычислить:

$$\sum_{i=1}^{100}\frac{1}{i^2}$$

Лабораторная работа №2

Массивы. Строки в C#

Задание. Написать программу на C# для решения следующих задач:

1. Дан одномерный массив, состоящий из N целочисленных элементов.

Ввести массив с клавиатуры.

Найти максимальный элемент.

Вычислить среднеарифметическое элементов массива.

Вывести массив на экран в обратном порядке.

2. Дан двумерный массив размерностью 4×6, заполненный целыми числами с клавиатуры. Сформировать одномерный массив, каждый элемент которого равен количеству элементов соответствующей строки, больших данного числа.

3. Дана строка, содержащая английский текст. Найти количество слов, начинающихся с буквы b.

Лабораторная работа №3

Функции. Файлы

Задание. Написать программу на C# для решения следующих задач:

1. Даны действительные числа s,t. Получить g(1.2,s) + g(t,s) - g(2s-1,st), где $g(a,b)=\frac{a^2-b^2}{2\cdot a\cdot b-a-b}+(a+b)\cdot\sqrt{\frac{|a+b|}{2}}$. Решение осуществить путем создания подпрограмм-функций.
2. Вычислить с использованием функции наименьшие элементы в строке и сумму номеров строк и столбцов, в которых они расположены, для матрицы A(10,15). Результаты формировать в одномерных массивах M(10) и S(10). Ввод и вывод массивов выполнить в отдельных функциях.
3. Дан файл f, элементы которого являются действительными числами. Найти модуль суммы и квадрат произведения элементов файла f.

Лабораторная работа №4

Классы. Указатель this

Задание. Написать программу на C# для решения следующей задачи:

Разработать класс: **Abiturient**: Фамилия, Имя, Отчество, Адрес, Оценки. Создать массив объектов. Вывести:

а) список абитуриентов, имеющих неудовлетворительные оценки;

б) список абитуриентов, сумма баллов у которых не меньше заданной;

в) выбрать N абитуриентов, имеющих самую высокую сумму баллов, и список абитуриентов, имеющих полупроходной балл.

Лабораторная работа №5

Перегрузка операций и методов

Задание. Написать программу на C# для решения следующих задач:

1. Для строки символов реализовать перегрузку операций:
   сравнение строк (операция ==);
   удаление из строки заданного символа (операция –).

Кроме того, членом класса сделать функцию с именем strset() для удаления из первой строки всех символов, встречающихся во второй строке.

2. Определить класс-строку. В класс включить два конструктора: для определения класса строки строкой символов и путем копирования другой строки (объекта класса строки). Определить операции над строками:

>> перевертывание строки (запись символов в обратном порядке);
++ нахождение наименьшего слова в строке.

Лабораторная работа №6

Наследование

Задание. Написать программу на C# для решения следующей задачи:

Требуется создать базовый класс «Транспортное средство» и определить общие и специфические методы для данного класса. Создать производные классы «Автомобиль», «Велосипед», «Повозка», в которые добавить свойства и методы. Часть методов переопределить. Создать массив объектов базового класса и заполнить объектами производных классов. Предусмотреть передачу аргументов конструкторам базового класса. Подсчитать время и стоимость перевозки пассажиров и грузов каждым транспортным средством.

Лабораторная работа №7

Виртуальные методы. Абстрактные классы

Задание. Написать программу на C# для решения следующих задач:

1. Разработать программу с использованием наследования классов, реализующую классы: графический объект, круг, квадрат. Используя виртуальные функции, выведите на экран размер и координаты графического объекта.
2. Создать абстрактный базовый класс уравнения с виртуальной функцией печать корней уравнения. Создать производные классы: линейное уравнение и квадратное уравнение. Определить функцию вычисления корней уравнений. Для проверки определить массив указателей на абстрактный класс, которым присваиваются адреса различных объектов.

Лабораторная работа №8

Обработка исключительных ситуаций

Задание. Написать программу на C# для решения следующих задач:

1. Создать три массива a, b и c размерами соответственно n1, n2 и n3 (n1≠n2≠n3). В массив a занести значения функции $f(x) = \ln(x-1), x \in [0;10], \Delta x = 0{,}5$ (при возникновении исключения заносить нули). Массив b заполнить случайными числами (среди них должны быть и отрицательные числа и нули). Массив c формируется следующим образом: $c_i=a_i+1/b_i$. Предусмотреть и обработать возникающие при этом исключительные ситуации (деление на ноль, корень из отрицательного числа, арифметическое переполнение, выход за пределы диапазона индексов массива и т.п.).
2. Проверить правильность перевода из двоичной системы счисления в троичную. Написать функцию, которая должна генерировать исключение некорректных значений параметров. Приведите пример некорректного использования.

Раздел 3. Самостоятельная работа

3.1. Проработка лекционного материала по темам лекций.
3.2. Подготовка к практическим работам по темам из Раздела 1.
3.3. Оформление отчетов по лабораторным работам.

Список литературы

1. Объектно-ориентированное программирование: Учебное пособие / Романенко В. В. - 2014. 475 с. [Электронный ресурс] - Режим доступа: https://edu.tusur.ru/publications/4872, дата обращения: 28.04.2018.
2. Павловская Т.А. C/C++. Программирование на языке высокого уровня : учебник для вузов / Т. А. Павловская. - СПб. : ПИТЕР, 2013. - 461 с.