МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

«ОБЧИСЛЮВАЛЬНА ТЕХНІКА ТА ПРОГРАМУВАННЯ»
Алгоритми та їх реалізація

Конспект лекцій

Рекомендовано Методичною радою КПІ ім. Ігоря Сікорського
як навчальний посібник для здобувачів ступеня бакалавра за освітньою програмою «Системи забезпечення споживачів електричної енергії», «Енергетичний менеджмент та енергоефективні технології» та «Інжиніринг інтелектуальних електротехнічних та мехатронних комплексів»
спеціальності 141 Електроенергетика, електротехніка та електромеханіка

Київ
КПІ ім. Ігоря Сікорського
2022

ОБЧИСЛЮВАЛЬНА ТЕХНІКА ТА ПРОГРАМУВАННЯ: Алгоритми та їх реалізація. Конспект лекцій [Електронний ресурс] : навч. посіб. для студ. спеціальності 141 Електроенергетика, електротехніка та електромеханіка / КПІ ім. Ігоря Сікорського; уклад.: Д. В. Філянін. – Електронні текстові дані (1 файл: 1,57 Мбайт). – Київ : КПІ ім. Ігоря Сікорського, 2022. – 99 с.

Гриф надано Методичною радою КПІ ім. Ігоря Сікорського (протокол № 5 від 26.05.2022 р.) за поданням Вченої ради НН ІЕЕ (протокол № 8 від 28.03.2022 р.)

Електронне мережне навчальне видання

«ОБЧИСЛЮВАЛЬНА ТЕХНІКА ТА ПРОГРАМУВАННЯ»
Алгоритми та їх реалізація

Конспект лекцій

Укладачі:	*Філянін Данило Володимирович*, канд. техн. наук, асистент.
Відповідальний редактор	*Попов В. А.*, д-р техн. наук, доцент
Рецензент	*Данілін А. В.*, канд. техн. наук, доцент кафедри АЕМК НН ІЕЕ КПІ ім. Ігоря Сікорського

Навчальний посібник з дисципліни «ОБЧИСЛЮВАЛЬНА ТЕХНІКА ТА ПРОГРАМУВАННЯ: Алгоритми та їх реалізація. Конспект лекцій» містить в собі лекційний матеріал з основ програмування на мові C#. Викладено принципи побудови алгоритмів і основи створення консольних додатків в середовищі Visual Studio.

Призначений для здобувачів ступеня бакалавра за освітньою програмою «Системи забезпечення споживачів електричної енергії», «Енергетичний менеджмент та енергоефективні технології», та «Інжиніринг інтелектуальних електротехнічних та мехатронних комплексів» спеціальності 141 Електроенергетика, електротехніка та електромеханіка.

Конспект лекцій з дисципліни

«Обчислювальна техніка та програмування»

кредитний модуль «Алгоритми та їх реалізація»

Навчальна дисципліна «Обчислювальна техніка та алгоритмічні мови» належить до обов'язкових дисциплін циклу природничо-наукової підготовки.

Предмет навчальної дисципліни: методи та алгоритми розв'язання інженерних задач за допомогою сучасних алгоритмічних мов з використанням сучасних комп'ютерних технологій.

Завдання курсу - навчитися писати програми і самостійно отримувати нові знання в області програмування.

Після вивчення цієї дисципліни ви повинні навчитися:

- використовувати сучасні методи програмування для вирішення

інженерних завдань у сфері енергопостачання;

- розробляти програмні проекти для комп'ютерів на основі використання технології об'єктно-орієнтованого програмування;
- створювати та застосовувати алгоритми для вирішення типових

задач обробки інформації.

Курс дисципліни складається з двох кредитних модулів:

1) Алгоритми та їх реалізація;
2) Об'єктно-орієнтоване програмування.

Перший розділ дисципліни «Алгоритми та їх реалізація» вивчається у цьому семестрі і має на меті навчити основам програмування типових прийомів обробки даних на мові C#. У лекційної частини розділу вивчаються елементи теорії алгоритмів і конструкції мови C#, прийоми перетворення математичних виразів у вигляд, зручний для програмування. Практичні навички програмування набуваються на практичних заняттях у спеціалізованих класах.

По завершенню першого кредитного модулю передбачений іспит.

Оцінка за семестр виставляється відповідно до рейтингової системи оцінювання. Ваш рейтинг з кредитного модуля складається з балів, які ви отримаєте за:

1) захист 14 практичних робіт;
2) дві відповіді на практичних заняттях;
3) одну модульну контрольну роботу;
4) одну розрахункову роботу;
5) відповідь на екзамені.

Основна література наведена у навчальному посібнику з виконання комп'ютерного практикуму.

Лекція 1. Поняття алгоритму, його властивості, базові елементи побудови алгоритмів. Мови програмування та сфери їх використання.

Алгоритми

Будь-яка людина щодня зустрічається з безліччю завдань від найпростіших і добре відомих до дуже складних. Для багатьох завдань існують певні правила (інструкції, розпорядження), що пояснюють виконавцю, як вирішувати дану задачу. Ці правила людина може вивчити заздалегідь або сформулювати сам в процесі рішення задачі. Чим точніше і зрозуміліше будуть описані правила вирішення завдань, тим швидше людина опанує ними і буде ефективніше їх застосовувати.

Вирішення багатьох завдань людина може передавати технічним пристроям - персональний комп'ютер, автоматам, роботам і т. д. Застосування технічних пристроїв пред'являє дуже суворі вимоги до точності опису правил і послідовності виконання дій. Тому розробляються спеціальні мови для чіткого і суворого опису різних правил.

Алгоритмізація це розділ інформатики, що вивчає методи і прийоми побудов алгоритмів, а також їх властивості.

Джерелами виникнення алгоритмів служать: спостереження, експеримент, наукова теорія, минулий досвід і ін.

Перед рішенням будь-якої задачі за допомогою персонального комп'ютера виконуються наступні етапи: постановка цієї задачі, побудова сценарію і алгоритмізація задачі.

На етапі постановки завдання описуються вихідні дані та передумови, формуються правила початку та закінчення рішення задачі (досягнення мети), тобто розробляється інформаційна або еквівалентна їй математична модель. Методом проб і помилок ведеться пошук методу розв'язання задачі (методу обчислень, методу перебору варіантів, методу розпізнавання). На підставі цього методу розробляється вихідний алгоритм, реалізація якого принципово можлива за допомогою персонального комп'ютера.

Алгоритмізація задачі - процес розробки (проектування) алгоритму

розв'язання задачі за допомогою персональний комп'ютерна основі її умови і вимог до кінцевого результату.

При розробці вихідного алгоритму і навіть при виборі моделі користувач, тобто людина, яка вирішує конкретну задачу, повинна мати уявлення про математичне забезпечення персонального комп'ютера.

Результатом алгоритмізації задачі є алгоритм.

Алгоритм - зрозуміле і точне розпорядження виконавцю здійснити послідовність дій, спрямованих на досягнення зазначеної мети або на вирішення поставленої задачі.

Алгоритм по відношенню до персонального комп'ютера - набір операцій і правил їх чергування, за допомогою якого, починаючи з деяких вихідних даних, можна вирішити завдання фіксованого типу.

Операція алгоритму - розпорядження про виконання окремої закінченої дії.

Термін алгоритм походить від імені узбецького вченого IX ст. Аль-Хорезмі, який у своїй праці "Арифметичний трактат", перекладеному в XII в. з арабської на латинь, виклав правила арифметичних дій над числами в позиційній десятковій системі числення. Ці правила і називали алгоритмами. Таким чином, правила додавання, віднімання, ділення, множення чисел, правила перетворення алгебраїчних виразів, правила побудови геометричних фігур, граматичні правила правопису слів і пропозицій - все це алгоритми. Багато правил та інструкцій, що записані в різних документах також можна віднести до алгоритмів.

Самі алгоритми залежно від мети, початкових умов завдання, шляхів її вирішення і визначеності дій виконавця підрозділяються на:

- детерміновані, або жорсткі;
- гнучкі алгоритми, наприклад, стохастичні.

Детермінований алгоритм задає певні дії, позначаючи їх в єдиній і

визначеній послідовності, забезпечуючи тим самим однозначний результат, якщо виконуються ті умови процесу чи завдання, для яких

розроблений алгоритм.

Імовірнісний (стохастичний) алгоритм дає програму рішення задачі кількома шляхами або способами, що призводять до ймовірного досягненню результату.

Евристичний алгоритм (від грецького слова "еврика" - "Я знайшов") - це такий алгоритм, у якому досягнення кінцевого результату програми дій однозначно не визначено, так само як не визначена вся послідовність дій виконавця. До евристичних алгоритмів відносять, наприклад, інструкції та приписи.

Часто, для отримання нових алгоритмів, використовуються вже існуючі алгоритми. Це здійснюється комбінуванням вже відомих алгоритмів або за допомогою еквівалентних перетворень алгоритмів.

Алгоритми називаються еквівалентними, якщо результати, одержані за допомогою цих алгоритмів для одних і тих же вихідних даних, однакові.

Типовий приклад еквівалентного перетворення алгоритмів - переклад з однієї алгоритмічного мови на іншу.

У загальному випадку алгоритмізація обчислювального процесу включає наступні дії:

- послідовну декомпозицію задачі, виділення автономних етапів;

обчислювального процесу і розбивку кожного етапу на окремі кроки;

- формальний запис змісту кожного етапу або кроку;
- визначення загального порядку виконання етапів або кроків;
- перевірку правильності алгоритму.

Послідовна декомпозиція передбачає поділ складного завдання на сукупність простіших підзадач.

Часто починаючі програмісти не приділяють етапу алгоритмізації достатньої уваги і навіть намагаються його ігнорувати. У результаті процес програмування сильно ускладнюється.

Значно простіше вирішувати завдання поступово, в два етапи (при цьому складність виконання кожного окремого етапу виходить в кілька разів менше

складності вихідної задачі).

На першому етапі треба намітити загальну стратегію вирішення завдання і скласти відповідний алгоритм. Причому для складної задачі алгоритмізація виконується поступово. Спочатку складається укрупнена схема рішення, потім - схеми роботи окремих блоків. Крім того, при алгоритмізації одного і того ж процесу можна використовувати кілька способів запису (починаючи з менш формалізованих форм).

На другому етапі залишається лише виконати кодування (програмування), замінивши формульно-словесні інструкції алгоритму операторами конкретної мови. Ця робота вже не пов'язана з великим розумовим напруженням. При нескладних задачах для її виконання досить знати загальні правила оформлення програм, правила опису даних, основні оператори (введення/виводу, обробки, управління).

Властивості алгоритмів

Властивості алгоритму - набір властивостей, що відрізняють алгоритм від будь-яких інших наборів дій і забезпечують його автоматичне виконання.

Алгоритми мають цілу низку властивостей: зрозумілість, дискретність, точність, результативність, масовість.

Зрозумілість для виконавця - наявність в алгоритмі тільки таких дій, які входять в систему команд виконавця, тобто алгоритм повинен бути заданий за допомогою таких вказівок, які виконавець (персональний комп'ютер, промисловий комп'ютер, контролер, однекристальна мікроЕОМ та ін.) може сприймати і виконувати до них необхідні дії (операції).

Дискретність - виконання команд алгоритму повинно бути послідовним, з точною фіксацією моментів закінчення виконання однієї команди і початку виконання наступної, тобто алгоритм повинен місити послідовність вказівок (команд), кожне з яких приводить до здійснення виконавцем одного кроку (дії).

Визначеність - кожне правило алгоритму має бути чітким, однозначним. Завдяки цій властивості виконання алгоритму носить механічний характер і не

вимагає ніяких додаткових вказівок або відомостей про розв язуваної задачі.

Результативність - алгоритм повинен забезпечувати можливість отримання результату після кінцевого числа кроків, тобто або завершатися вирішенням завдання після виконання алгоритму, або давати висновок про неможливість продовження рішення з якоїсь причини.

Масовість - означає, що алгоритм вирішення задачі розробляється в загальному вигляді, тобто він повинен бути застосовний для деякого класу завдань, які відрізняються лише вихідними даними. При цьому вихідні дані можуть вибиратися з деякої області, яка називається областю застосовності алгоритму.

Способи опису алгоритмів

Для чіткого завдання різних алгоритмів потрібно мати таку систему формальних позначень і правил, щоб сенс всякої послідовності дій трактувавсяточно і однозначно. Відповідні системи правил називають мовами опису алгоритмів.

Опис алгоритму залежить від виконавця - людини чи технічного засобу (комп'ютера).

Запис алгоритму для людини може мати звичайний текстуальний вигляд.

Для підвищення однозначності словесного опису вдаються до структуризації тексту відповідно до кроків алгоритмічного процесу. Коли потрібна висока наочність, застосовують графічну форму запису алгоритмів, наприклад, у вигляді блок- схем.

Таким чином, до засобів опису алгоритмів відносяться такі основні способи їх подання:

- словесний;
- графічний;
- псевдокод;
- табличний;
- програмний.

На практиці в якості виконавців алгоритмів використовуються комп'ютери чи інші обчислювальні пристрої (однокристальні мікроперсональний комп'ютер, промислові комп'ютери, технологічні контролери та ін.). Тому алгоритм, призначений для виконання на комп'ютері, повинен бути записаний на зрозумілій йому мові. І тут на перший план висувається необхідність точного запису команд, яка не залишає місця для довільного трактування їх виконавцем.

Мова для запису алгоритмів повинна бути формалізованою. Таку мова прийнято називати мовою програмування, а запис алгоритму цій мові - програмою для комп'ютера.

Отже, мова програмування - це формальна знакова система, призначена для запису комп'ютерних програм.

Мови програмування є формальними мовами, і в цьому їх принципова відмінність від природних мов. Природним мовам властивий ряд особливостей: неоднозначність (одне і те ж слово в залежності від контексту може мати різне значення), велика надмірність (наприклад, в ділових документах надмірна кількість символів досягає 90%), складність граматики й інші особливості. Зазначені особливості не дозволяють використовувати природні мови в якості мов програмування.

В даний час у світі існує кілька десятків мов програмування, які реально використовуються. У кожної з них є своя область застосування.

Наведемо приклади таких мов. За даними індексу Tiobe, що аналізує популярність мов програмування по всьому світу на основі згадування тієї чи

Aug 2016	Programming Language	Ratings
1	Java	19.010%
2	C	11.303%
3	C++	5.800%

іншої мови в популярних пошукових системах, таких як Google, Bing і Baidu, десятка мов з найвищим рейтингом така:

Aug 2016	Programming Language	Ratings
1	Java	19.010%
2	C	11.303%
3	C++	5.800%
4	C#	4.907%
5	Python	4.404%
6	PHP	3.173%
7	JavaScript	2.705%
8	Visual Basic .NET	2.518%
9	Perl	2.511%
10	Assembly language	2.364%
11	Delphi/Object Pascal	2.278%
12	Ruby	2.278%
13	Visual Basic	2.046%
14	Swift	1.983%
15	Objective-C	1.884%
16	Groovy	1.637%
17	R	1.605%
18	MATLAB	1.538%
19	PL/SQL	1.349%
20	Go	1.270%

Залежно від ступеня деталізації кроків алгоритму для комп'ютера, зазвичай визначається рівень мови програмування - чим менше деталізація, тим вище рівень мови.

За цим критерієм можна виділити такі рівні мов програмування:

- машинні;
- машинно-орієнтовані (мови асемблера);
- машинно-незалежні (мови високого рівня).

Машинні та машинно-орієнтовані мови - це мови низького рівня, що вимагають вказівки дрібних деталей процесу обробки даних. Мови ж високого рівня імітують природні мови, використовуючи деякі слова розмовної мови і загальноприйняті математичні символи. Ці мови більш зручні для людини.

Мови високого рівня поділяються на:

- процедурні (алгоритмічні) (Basic, Pascal, C та ін.), які призначені для однозначного опису алгоритмів; для вирішення завдання процедурні мови вимагають в тій чи іншій формі явно виписати процедуру її розв'язання;

- логічні (Prolog, Lisp та ін.), які орієнтовані не на розробку алгоритму розв'язання задачі, а на систематичний і формалізований опис завдання з тим, щоб рішення випливало з складеного опису;
- об'єктно орієнтовані (C ++, C #, Java, Object Pascal, та ін.), в основі яких лежить поняття об'єкта, який поєднує у собі дані і дії над ними. Програма на об'єктно-орієнтованій мові, вирішуючи деяку задачу, по суті, описує частина світу, що відноситься до цього завдання.

Основні переваги алгоритмічних мов програмування перед машинними та машинно-орієнтованими мовами такі:

- алфавіт алгоритмічної мови значно ширше алфавіту машинної мови, що істотно збільшує наочність тексту програми;
- набір операцій, допустимих для використання, не залежить від набору машинних операцій, а вибирається з міркувань зручності формулювання алгоритмів розв'язання задач певного класу;
- формат операторів досить гнучкий і зручний для використання, що дозволяє за допомогою одного речення задати досить змістовний етап обробки даних;
- необхідні операції задаються за допомогою загальноприйнятих математичних позначень;
- даним в алгоритмічних мовах присвоюються індивідуальні імена, обрані програмістом;
- в мові може бути передбачений значно ширший набір типів даних порівняно з набором машинних типів даних.

Алгоритмічні мови програмування значною мірою є машинно-незалежними. Вони полегшують роботу програміста і підвищують надійність створюваних програм.

Компоненти алгоритмічної мови

В даному курсі ви будете вивчати мову програмування C#. Це об'єктно-орієнтована мова програмування, розроблена в 1998-2001 роках групою інженерів під керівництвом Андерса Хейлсберга в компанії Microsoft як мова

розробки додатків для платформи Microsoft .NET Framework. (Microsoft .NET — програмна технологія, запропонована фірмою Microsoft як платформа для створення як звичайних програм, так і веб-застосунків.)

Синтаксис C# близький до C++ і Java. На сьогодні C# є флагманською мовою корпорації Microsoft для програмування в середовищі Windows.

Visual Studio 2015

MonoDevelop — відкрите інтегроване середовище розробки для платформ Linux, Mac OS X[1] та Microsoft Windows[2], передусім націлене на розробку програм, які використовують і Mono, і Microsoft .NET framework. На даний момент підтримуються мови C#, Java, Boo, Visual Basic.NET, CIL, Python, Vala, C та C++. Також MonoDevelop підтримує такі технології, як Gtk#, ASP.NET MVC, Silverlight, MonoMac и MonoTouch.

SharpDevelop — свободная среда разработки для C#, Visual Basic .NET, Boo, IronPython, IronRuby, F#, C++.

Xamarin Studio - для розробки мобільних застосувань на мові C# з використанням технологій .NET.

Лекція 2. Базові елементи мови програмування C#. Операції консольного введення і виведення. Типи даних.

Програма записана певною мовою називається вихідним текстом і зазвичай зберігається в текстовому файлі.

Для перекладу програми на мову низького рівня, зрозумілу комп'ютеру, існують спеціальні програми - компілятори. Результатом роботи компілятора (іншими словами, результатом процесу компіляції) є виконуваний код, який при програмуванні в середовищі Windows записується у файл з розширенням .exe.

Компілятор перед створенням виконуваного коду перевіряє синтаксис програми, що подається йому для перекладу. Пошуком же семантичних помилок займається програміст в процесі тестування і налагодження своєї програми.

Налагодження - це пошук і виправлення помилок у програмі. Тестування - це складання спеціальних наборів вхідних і вихідних даних (тестів), а потім виконання програми і перевірка отриманих результатів в пошуках можливих семантичних або логічних помилок.

Мову програмування C# (як і будь-який інший мову) утворюють три складові: алфавіт, синтаксис і семантика.

Алфавіт - це фіксований для даної мови набір основних символів, тобто "Букв алфавіту", з яких повинен складатися будь-який текст на цій мові, - ніякі інші символи у тексті не допускаються.

Синтаксис - це правила побудови фраз, що дозволяють визначити, правильно чи неправильно написана та чи інша фраза. Точніше кажучи, синтаксис мови являє собою набір правил, що встановлюють, які комбінації символів є осмисленими виразами на цій мові.

Семантика визначає смислове значення фраз мови. Будучи системою правил тлумачення окремих мовних конструкцій, семантика встановлює, які послідовності дій описуються тими чи іншими фразами мови і в кінцевому підсумку який алгоритм визначений даним текстом на алгоритмічній мові.

Алфавіт мови C# складається з букв, цифр та спеціальних символів. В якості букв використовуються прописні букви латинського та національних алфавітів, наприклад від A до Z і рядкові від a до z, а також знак підкреслення _. В якості десяткових цифр використовуються арабські цифри від 0 до 9. Спеціальні символи в мові C# застосовуються для різних цілей: від організації тексту програми до визначення вказівок компілятору. Спеціальні символи перераховані в табл. 1.

Таблиця 1

Символ	Найменування	Символ	Найменування	Символ	Найменування
,	Кома	<	Менше	%	Відсоток
.	Точка	>	Більше	&	Амперсант
;	Крапка з комою	[	Ліва квадратна дужка	^	Кришка
:	Двокрапка	]	Права квадратна дужка	-	Мінус
?	Знак питання	!	Знак оклику	=	Знак рівності
‘	Одиночна лапка (апостроф)	|	Вертикальна риса	+	Плюс
(	Ліва кругла дужка	/	Похила риса вправо (прямий слеш)	*	Зірочка
)	Права кругла дужка	\	Похила риса вліво (зворотний слеш)	“	Подвійна лапка
{	Ліва фігурна дужка	~	Тільда		
}	Права фігурна дужка	#	Знак номера		

Лексема -це мінімальна одиниця мови, що має самостійний сенс. Існують наступні види лексем:

- імена (ідентифікатори);
- ключові слова;

□ знаки операцій;

□ роздільники;

□ літерали (константи).

Лексеми мови програмування аналогічні словами природної мови. Наприклад, лексемами є число 128 (але не його частина 12), ім'я Vasia, ключове слово for і знак операції додавання +. Далі ми розглянемо лексеми докладніше.

Коментарі призначені для запису пояснень до програми і формування документації. Правила запису коментарів описані далі в цьому розділі.

З лексем складаються вирази й оператори. Вираз задає правило обчислення деякого значення. Наприклад, вираз a + b задає правило обчислення суми двох величин.

Ідентифікатори використовуються як імена змінних, функцій і типів даних. Вони записуються за такими правилами:

• Ідентифікатори починаються з літери (знак підкреслення також є буквою).

• Ідентифікатор може складатися з латинських букв і цифр пробіли, точки та **інші** спеціальні символи при написанні ідентифікаторів недопустимі).

• Між двома ідентифікаторами повинен бути, принаймні, хоча б один пробіл.

При написанні ідентифікаторів можна використовувати як прописні, так і малі літери. На відміну від інших мов програмування, компілятор мови C# розрізняє їх в записі ідентифікатора.

Для поліпшення читабельності програми слід давати об'єктам осмислені імена, складені відповідно до певних правил. Зрозумілі й узгоджені між собою імена - основа гарного стилю програмування. Існує кілька видів так званих нотацій - угод про правила створення імен.

У нотації Паскаля кожне слово, яке складає ідентифікатор, починається з великої літери, наприклад, MaxLength, MyFuzzyShooshpanchik.

Угорська нотація (її запропонував угорець за національністю, співробітник компанії Microsoft) відрізняється від попередньої наявністю префікса, відповідного типу величини, наприклад, iMaxLength, lpfnMyFuzzyShooshpanchik.

Згідно нотації Camel, з великої літери починається кожне слово, яке складає ідентифікатор, крім першого, наприклад, maxLength, myFuzzyShooshpanchik. Людині з багатою фантазією абрис імені може нагадувати верблюда, звідки і пішла назва цієї нотації.

Ще одна традиція - розділяти слова, складові ім'я, знаками підкреслення: maxjength, my_fuzzy_shooshpanchik, при цьому всі складові частини починаються з малої літери.

УС# для іменування різних видів програмних об'єктів найчастіше використовуються дві нотації: Паскаля і Camel.

Ключові слова

Ключові слова - це зарезервовані ідентифікатори, які мають спеціальне значення для компілятора. їх можна використовувати тільки в тому сенсі, в якому вони визначені.

Знаки операцій і роздільники

Знак операції - це один або більше символів, що визначають дію над операндами. Усередині знака операції пробіли не допускаються. Наприклад, у виразі a + = b знак + = є знаком операції, a a і b - операндами. Символи, складові знак операцій, можуть бути як спеціальними, наприклад, &&, | і <, так і літерними, такими як as або new.

Літерали

Літералами, або константами, називають незмінні величини. У C# є логічні, цілі, речові, символьні і рядкові константи, а також константа null. Компілятор, виділивши константу в якості лексеми, відносить її до одного з типів даних по її зовнішньому вигляду. Програміст може задати тип константи і самостійно

Опис і приклади констант кожного типу наведено в табл. 2.2. Приклади,

що ілюструють найбільш часто вживані форми констант, виділені напівжирним шрифтом (при першому читанні можна звернути увагу тільки на них).

Таблица 2.2. Константы в C#

Константа	Описание	Примеры
Логическая	`true` (истина) или `false` (ложь)	`true` `false`
Целая	*Десятичная*: последовательность десятичных цифр (0, 1, 2, 3, 4, 5, 6, 7, 8, 9), за которой может следовать суффикс (U, u, L, l, UL, Ul, uL, ul, LU, Lu, lU, lu)	`8 0 199226` `8u 0Lu 199226L`

	Шестнадцатеричная: символы `0x` или `0X`, за которыми следуют шестнадцатеричные цифры (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F), а за цифрами, в свою очередь, может следовать суффикс (U, u, L, l, UL, Ul, uL, ul, LU, Lu, lU, lu)	`0xA 0x1B8 0X00FF`
		`0xAU 0x1B8LU`
		`0X00FFl`

Вещественная	*С фиксированной точкой*[1]:
	[цифры][.][цифры][суффикс]
	Суффикс — один из символов F, f, D, d, M, m
		`5.7 .001 35`
		`5.7F .001d 35`
		`5F .001f 35m`

	С порядком:
	[цифры][.][цифры]{E|e}[+|–][цифры][суффикс]
	Суффикс — один из символов F, f, D, d, M, m
		`0.2E6 .11e+3 5E-10`
		`0.2E6D .11e-3`
		`5E10`

Символьная	Символ, заключенный в апострофы	`'A' 'ю' '*'`
		`'\0' '\n'`
		`'\xF' '\x74'`
		`'\uA81B'`

Строковая	Последовательность символов, заключенная в кавычки	`"Здесь был Vasia"`
		`"\t3начение r = \xF5 \n"`
		`"Здесь был \u0056\u0061"`
		`"C:\\temp\\file1.txt"`
		`@"C:\temp\file1.txt"`

Константа `null`	Ссылка, которая не указывает ни на какой объект	`null`

Типи даних

Дані, з якими працює програма, зберігаються в оперативній пам'яті. Природно, що компілятору необхідно точно знати, скільки місця вони

займають, як саме закодовані і які дії з ними можна виконувати. Все це здасться при описі даних за допомогою типу.

Тип даних однозначно визначає:

- внутрішнє подання даних, а отже, і безліч їх можливих значень;
- допустимі дії над даними (операції і функції).

Наприклад, цілі і дійсні числа, навіть якщо вони займають однаковий обсяг пам'яті, мають зовсім різні діапазони можливих значень; цілі числа можна множити один на одного, а, наприклад, символи - не можна.

Кожен вираз в програмі має певний тип. Величин, що не мають ніякого типу, не існує. Компілятор використовує інформацію про тип при перевірці допустимості описаних в програмі дій.

Основними даними, що обробляються програмою, - це змінні і константи.

Змінні - це дані, які можуть змінювати свої значення в процесі виконання програми.

Всі змінні в мові C# повинні бути описані явно. Це означає, що, по-перше, на початку кожної програми або функції Ви повинні привести список імен (ідентифікаторів) всіх використовуваних змінних, а по-друге, вказати тип кожної з них.

Оператор опису складається з назви типу і списку імен змінних, розділених комою. Наприкінці обов'язково повинна стояти крапка з комою. При описі можливе завдання початкового значення змінної.

Ім'я змінної - є ідентифікатором - тобто будь-яка послідовність великих і малих літер англійського алфавіту, цифр і символу підкреслення '_'.

Подібно будь-якій іншій мові програмування, в C# визначені ключові слова для фундаментальних типів даних, які застосовуються для опису змінних. Однак на відміну від інших мов програмування, в C# ці ключові слова являють собою щось більше, ніж просто розпізнавані компілятором лексеми. По суті, вони є скороченими позначеннями повноцінних типів з простору імен System.

Перерахуємо ці системні типи даних разом з їх діапазонами значень та відповідними ключовими словами C#

Таблица 3.4. Внутренние типы данных C#

Сокращенное обозначение в C#	Совместимость с CLS	Системный тип	Диапазон	Описание
bool	Да	System.Boolean	true или false	Признак истинности или ложности
sbyte	Нет	System.SByte	от –128 до 127	8-битное число со знаком
byte	Да	System.Byte	от 0 до 255	8-битное число без знака
short	Да	System.Int16	от –32 768 до 32 767	16-битное число со знаком
ushort	Нет	System.UInt16	от 0 до 65535	16-битное число без знака
int	Да	System.Int32	от –2147483648 до 2147483647	32-битное число со знаком
uint	Нет	System.UInt32	от 0 до 4294967295	32-битное число без знака
long	Да	System.Int64	от –9223372036854775808 до 9223372036854775807	64-битное число со знаком
ulong	Нет	System.UInt64	от 0 до 18446744073709551615	64-битное число без знака
char	Да	System.Char	от U+0000 до U+ffff	Одиночный 16-битный символ Unicode
float	Да	System.Single	от -3.4×10^{38} до $+3.4\times10^{38}$	32-битное число с плавающей точкой
double	Да	System.Double	от $\pm5.0\times10^{-324}$ до $\pm1.7\times10^{308}$	64-битное число с плавающей точкой
decimal	Да	System.Decimal	(от -7.9×10^{28} до 7.9×10^{28}) / ($10^{\text{от 0 до 28}}$)	128-битное число со знаком
string	Да	System.String	Ограничен объемом системной памяти	Набор символов Unicode
object	Да	System.Object	В переменной object может храниться любой тип данных	Базовый класс для всех типов в мире .NET

Для оголошення локальної зміною необхідно вказати тип даних і відразу за ним ім'я змінної. Щоб подивитися, як це виглядає, створимо новий проект консольного застосування по імені BasicDataTypes і модифікуємо клас Program так, щоб у його методі Main() викликався наступний допоміжний метод:

```
static void LocalVarDeclarations ()
{
Console.WriteLine ("=> Data Declarations:");
// Локальные переменные объявляются следующим образом:
// типДанных. имяПеременной;
int myint;
string myString;
Console.WriteLine();
}
```

Так звана область дії змінної, тобто область програми, де можна використовувати змінну, починається в точці її опису і триває до кінця блоку, усередині якого вона описана. Блок - це код, укладений у фігурні дужки. Основне призначення блоку - угруповання операторів. У C# будь-яка змінна описана усередині якого-небудь блоку: класу, методу чи блоку всередині методу.

Ім'я змінної повинно бути унікальним в області її дії. Область дії поширюється на вкладені в метод блоки, з цього випливає, що у вкладеному блоці не можна оголосити змінну з таким же їм ям, що і в охоплює його, наприклад:

Використання локальної змінної до присвоювання їй початкового значення призведе до помилки на етапі компіляції. З урахуванням цього, рекомендується привласнювати початкові значення локальних змінних під час їх оголошення. Робити це можна як в одній і тій же рядку, так і розносити оголошення і присвоювання на два окремих оператора коду.

```
static void LocalVarDeclarations ()
{
Console.WriteLine ("=> Data Declarations:");
// Локальные переменные объявляются и инициализируются следующим образом:
// типДанных имяПеременной = начальноеЗначение;
int mylnt = 0;
// Объявлять и присваивать можно также в двух отдельных строках.
string myString;
myString = "This is my character data";
Console.WriteLine();
}
```

Також дозволено оголошувати безліч змінних одного і того ж типу в одному рядку коду, як показано нижче на прикладі трьох змінних bool:

```
static void LocalVarDeclarations ()
{
Console.WriteLine ("=> Data Declarations:");
int mylnt = 0; string myString;
myString = "This is my character data";
// Объявить три переменных типа bool в одной строке.
bool b1 = true, b2 = false, b3 = b1;
```

```
Console.WriteLine();
}
```

Числові типи .NET підтримують властивості **MaxValue HMinValue,** які надають інформацію про діапазоні значень, що зберігаються в конкретному типі. Крім властивостей **MinValue** і **MaxValue, кожен** числовий тип може визначати додаткові корисні члени. Наприклад, тип **System.Double** дозволяє отримувати значення для епсилон і нескінченності (представляють інтерес для тих, хто займається вирішенням математичних задач). З метою ілюстрації розглянемо такий код:

```
static void DataTypeFunctionality()
{
Console.WriteLine("=> Data type Functionality:");
Console.WriteLine("Max of int: {0}", int.MaxValue);
Console.WriteLine("Min of int: {0}", int.MinValue);
Console.WriteLine("Max of double: {0}", double.MaxValue);
Console.WriteLine("Min of double: {0}", double.MinValue);
Console.WriteLine("double.Epsilon: {0}", double.Epsilon);
Console.WriteLine("double.Positivelnfinity: { 0}",
double.Positivelnfinity);
Console.WriteLine("double.Negativelnfinity: {0}",
double.Negativelnfinity);
Console.WriteLine();
}
```

Розглянемо структуру найпростішої програми на C#

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace MyFirstProject
{
	class Program
	{
		static void Main(string[] args)
		{

		}
	}
}
```

Перші рядки містять директиву using, яка повідомляє компілятору, де він повинен шукати класи (типи), не визначені в даному просторі імен. За замовчуванням вказано стандартний простір імен System, де визначена основна частина типів середовища .NET.

Наступний рядок

```
namespace MyFirstProject
```

визначає простір імен додатку. За замовчаннями як ім'я простору імен

вибирається ім'я проекту. Область дії простору імен визначається блоком коду між фігурними дужками. Простір імен забезпечує спосіб відокремлення одного набору імен від іншого. Імена, оголошені в одному просторі імен, не конфліктують з іменами, оголошеними в іншому просторі імен.

Слово class, розташоване в першому рядку тексту першої програми, відноситься до об'єктно орієнтованої частини мови, і буде детально розглянуто пізніше. Слово class повинне бути присутнім у будь-якій програмі на C# хоча б один раз.

Фраза static void Main() є заголовком методу Main. Наступний блок фігурних дужок обмежує тіло методу Main. Ім'я методу Main не може бути змінено, оскільки система саме з цієї підпрограми починає виконання додатку (так звана точка входу).

Майже у всіх прикладах додатків, створюваних протягом перших кількох розділів, буде інтенсивно використовуватися клас System.Console. І хоча насправді консольний користувальницький інтерфейс не настільки привабливий, як графічний користувальницький інтерфейс або інтерфейс веб-додатки, обмеження початкових прикладів до консольних програм дозволяє зосередити всю увагу на синтаксисі C # і ключових аспектах платформи .NET, не відволікаючись на складні деталі побудови графічних користувацьких інтерфейсів настільних додатків або веб-сайтів.

Тип **Console** визначає методи, які дозволяють виконувати консольне введення і виведення**.** Метод **WriteLine** () дозволяє помістити в потік виводу рядок тексту (включаючи символ повернення каретки). Метод **Write ()** поміщає в потік виводу текст без символу повернення каретки. Метод **ReadLine** () дозволяє отримати інформацію з потоку введення аж до натискання клавіші <Enter>. Метод **Read** () використовується для захоплення з потоку введення одиночного символу.

Для демонстрації базового введення-виведення із застосуванням класу **Console** запишемо такий код. Наприклад, ми могли б запросити у користувача його їм я і вік (який для простоти буде трактуватися як текстове значення, а не

звичне числове):

```
class Program {
static void Main(string[] args)
{
Console.Write("Please enter your name: ");
// Запрос на ввод имени
string userName = Console.ReadLine ();
Console.Write("Please enter your age: ");
// Запрос на ввод возраста
string userAge = Console.ReadLine ();
// Изменить цвет переднего плана, просто ради интереса.
ConsoleColor prevColor = Console.ForegroundColor;
Console.ForegroundColor = ConsoleColor.Yellow;
// Вывести полученные сведения на консоль.
Console.WriteLine("Hello {0}! You are {1} years old.", userName, userAge);
// Восстановить предыдущий цвет переднего плана.
Console.ForegroundColor = prevColor;
Console.ReadLine();
}
}
```

При виведенні рядку з сегментами даних, значення яких залишаються невідомими до етапу виконання, всередині нього допускається вказувати заповнювач, використовуючи синтаксис у вигляді фігурних дужок. Під час виконання на місце кожного такого заповнювача підставляється передане в Console .WriteLine() значення.

Перший параметр методу WriteLine () є строковим літералом, який містить заповнювачі виду {0}, {1}, {2} і т.д. Порядкові числа заповнювачів у фігурних дужках завжди починаються з 0. Інші параметри WriteLine() - це просто значення, які повинні підставлятися на місці заповнювачів.

Один і той же заповнювач може повторюватися в межах заданої рядки. Наприклад, для створення рядка "9, Number9, Number9м можна було б написати такий код:

```
// Вывод строки "9, Number9, Number9"
Console.WriteLine("{0}, Number {0}, Number {0}", 9);
```

Кожен заповнювач допускається розміщувати в будь-якому місці всередині строкового літерала, і не обов'язково, щоб заповнювачі вказувалися в зростаючій послідовності своїх номерів, наприклад:

```
// Выводит: 20, 10, 30
Console.WriteLine ("{1}, {0}, {2}", 10, 20, 30);
```

Якщо для числових даних потрібно більш складне форматування, кожен заповнювач може додатково містити різноманітні символи форматування. Перерахуємо найбільш корисні з них.

Таблица 3.3. Символы для форматирования числовых данных в .NET

Символ форматирования	Описание
C или c	Используется для форматирования денежных значений. По умолчанию значение предваряется символом локальной валюты (например, знаком доллара ($) для культуры US English)
D или d	Используется для форматирования десятичных чисел. В этом флаге можно также указывать минимальное количество цифр для представления значения
E или e	Используется для экспоненциального представления. Регистр этого флага указывает, в каком регистре должна представляться экспоненциальная константа — в верхнем (E) или в нижнем (e)
F или f	Используется для форматирования с фиксированной точкой. В этом флаге можно также указывать минимальное количество цифр для представления значения
G или g	Означает *general* (общий). Этот флаг может использоваться для представления чисел в формате с фиксированной точкой или экспоненциальном формате
N или n	Используется для базового числового форматирования (с запятыми)
X или x	Используется для шестнадцатеричного форматирования. В случае символа X в верхнем регистре шестнадцатеричное представление будет содержать символы верхнего регистра

Ці символи форматування додаються у вигляді суфіксів до заповнювачів після двокрапки (наприклад, {0: З}, {1: d}, {2: X}).

З метою ілюстрації запишемо такий код

```
// Использовать несколько дескрипторов формата
static void FormatNumericalData ()
{
Console.WriteLine ("The value 99999 in various formats:");
Console.WriteLine ("c format: {0:c}", 99999);
Console.WriteLine("d9 format: {0:d9}", 99999);
Console.WriteLine("f3 format: {0:f3}", 99999);
Console.WriteLine("n format: {0:n}", 99999);
// Обратите внимание, что использование верхнего или нижнего регистра для x
// определяет, в каком регистре отображаются символы в шестнадцатеричном формате
Console.WriteLine ("E format: {0:E}", 99999);
Console.WriteLine("e format: {0:e}", 99999);
```

Розглянемо найпростіші способи введення з клавіатури. У класі Console визначені методи введення рядка і окремого символу, але немає методів, які дозволяють безпосередньо зчитувати з клавіатури числа. Введення числових даних виконується в два етапи:

1. Символи, що представляють собою число, вводяться з клавіатури в рядкову змінну.

2. Виконується перетворення з рядка в змінну відповідного типу. Перетворення можна виконати або за допомогою спеціального класу Convert, визначеного в просторі імен System, або за допомогою методу Parse, наявного

в кожному стандартному арифметичному класі.

```
using System;
namespace ConsoleReadTest {
class Program {
static void Main(string[] args)
{
Console.WriteLine("Введіть рядок");
string s = Console.ReadLine();
Console.WriteLine("s =" + s);
Console.WriteLine("Введіть символ");
char c = (char) Console.Read();
Console.ReadLine();
Console.WriteLine("c =" + c);
string buf; // рядок - буфер для введення чисел
Console.WriteLine("Введіть ціле число");
buf = Console.ReadLine();
int i = Convert.ToInt32(buf);
Console.WriteLine(i);
Console.WriteLine("Введіть дійсне число");
buf = Console.ReadLine();
double x = Convert.ToDouble(buf);
Console.WriteLine(x);
Console.WriteLine("Введіть дійсне число");
buf = Console.ReadLine();
double y =double.Parse(buf);
Console.WriteLine(y);
Console.WriteLine("Введіть дійсне число");
buf = Console.ReadLine();
decimal z = decimal.Parse(buf);
Console.WriteLine(z);
}
}
}
using System;
namespace ConsoleReadTest {
class Program {
static void Main(string[] args)
{
Console.WriteLine("Введіть рядок");
var s = Console.ReadLine();
Console.WriteLine("s =" + s);
Console.WriteLine("Введіть символ");
var c = (char)Console.Read();
Console.ReadLine();
Console.WriteLine("c =" + c);
Console.WriteLine("Введіть ціле число");
var buf = Console.ReadLine();
var i = Convert.ToInt32(buf);
Console.WriteLine(i);
Console.WriteLine("Введіть дійсне число");
buf = Console.ReadLine();
var x = Convert.ToDouble(buf);
Console.WriteLine(x);
Console.WriteLine("Введіть дійсне число");
buf = Console.ReadLine();
var y =double.Parse(buf);
Console.WriteLine(y);
Console.WriteLine("Введіть дійсне число");
buf = Console.ReadLine();
var z = decimal.Parse(buf);
Console.WriteLine(z);
```

```
}
}
}
```

До цього прикладу необхідно зробити кілька пояснень. Введення рядка виконується в операторі 1. Довжина рядка не обмежена, введення виконується до символу перекладу рядка.

Введення символу виконується за допомогою методу Read, який зчитує один символ з вхідного потоку (оператор 2). Метод повертає значення типу int, що представляє собою код символу, або -1, якщо символів у вхідному потоці немає (наприклад, користувач натиснув клавішу Enter). Оскільки нам потрібно не int, а char, а неявного перетворення від int до char не існує, доводиться застосувати операцію явного перетворення типу, яка описана в розділі «Явна перетворення типу» (див. С. 49).

За оператором 2 записаний оператор 3, який зчитує залишок рядка і нікуди його не передає. Це необхідно тому, що введення даних виконується через буфер - спеціальну область оперативної пам'яті. Фактично, дані спочатку заносяться в буфер, а потім зчитуються звідти процедурами введення. Занесення в буфер виконується після натискання клавіші Enter разом з її кодом. Метод Read, на відміну від ReadLine, що не очищає буфер, тому наступний після нього введення буде виконуватися з того місця, на якому закінчився попередній.

У операторах 4 і 5 використовуються методи класу Convert, в операторах 6 і 7 - методи Parse класів Double і Decimal бібліотеки .NET, які використовуються тут через імена типів C # double і decimal.

Лекція 3. Вирази та арифметичні оператори Вираз - це правило обчислення значення.

У виразі беруть участь операнди, об'єднані знаками операцій. Операндами найпростішого вирази можуть бути константи, змінні і виклики функцій.

Наприклад, *a*+2 - це вираз, в якому + є знаком операції, а *a* і 2 - операндами. Пробіл всередині знака операції, що складається з декількох символів, не допускаються.

За кількістю що беруть участь в одній операції операндів операції діляться на унарні, бінарні і тернарні.

Таблиця 3.1. Операції C #

Категорія	Знак операції	Назва
Первинні		Доступ до елементу
	x ()	Виклик методу або делегата
	x []	Доступ до елементу
	X++	Постфіксний інкремент
	X--	Постфіксний декремент
	new	Виділення пам'яті
	typeof	Отримання типу
	checked	Перевіряється код
	unchecked	Непроверяем код
Унарні	+	Унарний плюс
	-	Унарний мінус (арифметичне заперечення)
	!	Логічне заперечення
	~	Порозрядне заперечення
	++X	Префіксний інкремент
	--X	Префіксний декремент
	(тип) x	Перетворення типу
Мультиплікативні	*	Множення
	/	Ділення
	%	Залишок від ділення
Адитивні	+	Додавання
	-	Віднімання
Зсуву	<<	Зсув вліво
	>>	Зсув вправо
Відношення та перевірки типу	<	Менше
	>	Більше
	<=	Менше або дорівнює
	>=	Більше або дорівнює

	is as	Перевірка приналежності типом Приведення типу
Перевірки на рівність	==	дорівнює
	!=	Не дорівнює
Порозрядні логічні	&	Поразрядна кон'юнкція (І)
	^	Порозрядне виключаюче АБО
	|	Поразрядна диз'юнкція (АБО)
Умовні логічні	&&	Логічне І
	||	Логічне АБО
Умовний	?	Умовна операція
Присвоювання	= *= /=	Присвоєння Множення з присвоєнням Ділення з присвоєнням
	%= +=	Залишок відділення з присвоєнням Додавання з присвоєнням
	-=	Віднімання з присвоєнням
	<<=	Зсув вліво з присвоєнням
	>>=	Зсув вправо з присвоєнням
	&=	Порозрядне І з присвоєнням
	^=	Порозрядне виключає АБО з присвоєнням
	|=	Порозрядне АБО з присвоєнням

Операції у виразі виконуються в певному порядку відповідно до пріоритетів, як і в математиці.

У табл. 3.1 операції розташовані за спаданням пріоритетів, рівні пріоритетів розділені в таблиці горизонтальними лініями. Результат обчислення виразу характеризується значенням і типом. Наприклад, нехай a і b - змінні цілого типу і описані так:

```
int a = 2;
int b = 5;
```

Тоді вираз a + b має значення 7 і тип int a вираз a - b має значення, рівне вміщеної в змінну a (в даному випадку - 5), і тип, що співпадає з типом цієї змінної.

Для зміни порядку виконання операцій використовуються круглі дужки, рівень їх вкладеності практично не обмежений.

Наприклад, a + b + c означає (a + b) + c, а a - b - c означає a - (b - c). Тобто

спочатку обчислюється вираз b - c, а потім його результат стає правим операндом для операції присвоювання змінної a.

При обчисленні виразів може виникнути необхідність у перетворенні типів. Якщо операнди, що входять у вираз, одного типу і операція для цього типу визначена, то результат виразу буде мати той же тип.

Якщо операнди різного типу або операція для цього типу не визначена, перед обчисленнями автоматично виконується перетворення типу за правилами, що забезпечує приведення коротших типів до більш довгим для збереження значущості і точності. Автоматичне (неявне) перетворення можливо не завжди, а тільки якщо при цьому не може трапитися втрата значущості.

Якщо неявного перетворення з одного типу в інший не існує, програміст може задати явне перетворення типу за допомогою операції (тип) x.

Інкремент і декремент

Операції інкремента (++) і декремента, що називають також операціями збільшення та зменшення на одиницю, мають дві форми запису - Префіксний, коли знак операції записується перед операндом, і постфіксними.

У префиксной формі спочатку змінюється операнд, а потім його значення стає результуючим значенням виразу, а в постфіксной формі значенням виразу є початкове значення операнда, після чого він змінюється.

Операції заперечення

Арифметичне заперечення (унарний мінус -) змінює знак операнда на протилежний.

Стандартна операція заперечення визначена для типів int, long, float, double і decimal. До величин інших типів її можна застосовувати, якщо для них можливо неявне перетворення до цих типів. Тип результату відповідає типу операції.

Логічне заперечення (!) Визначено для типу bool. Результат операції - значення false, якщо операнд дорівнює true, і значення true, якщо операнд дорівнює false.

Порозрядне заперечення (-), часто зване побітовим, інвертує кожен розряд в двійковому поданні операнда типу int, uint, long або ulong.

Явне перетворення типу

Операція використовується, як і випливає з її назви, для явного перетворення величини з одного типу в інший. Це потрібно в тому випадку, коли неявного перетворення не існує. При перетворенні з довшого типу в більш короткий можлива втрата інформації, якщо початкове значення виходить за межі діапазону результуючого типу .

Формат операції:

(Тип) вираз

Тут тип - це їм я того типу, в який здійснюється перетворення, а вираз найчастіше представляє собою ім'я змінної, наприклад:

```
long b = 300:
int a = (int) b: // дані не втрачаються
byte d = (byte) a: // дані губляться
```

Множення, ділення і залишок від ділення

Операція множення (*) повертає результат перемноження двох операндів. Стандартна операція множення визначена для типів int, uint, long, ulong, float, double і decimal.

Операція поділу (/) обчислює частка від ділення першого операнда на другий. Стандартна операція ділення визначена для типів int, uint, long, ulong, float, double і decimal.

Операція залишку від ділення (%) також інтерпретується по-різному для цілих, речових і фінансових величин. Якщо обидва операнда цілочисельні, результат операції обчислюється за формулою x - (x / y) * y.

Додавання і віднімання

Операція додавання (+) повертає суму двох операндів. Стандартна операція додавання визначена для типів int, u1nt, long, ulong, float, double і decimal.

Операція віднімання (-) повертає різницю двох операндів. Стандартна операція віднімання визначена для типів int, uint, long, ulong, float, double і

decimal.

Операції зсуву

Операції зсуву (<< та >>) застосовуються до цілочисельних операндів. Вони зсувають двійкове представлення першого операнда вліво або вправо на кількість двійкових розрядів, задане друге операндом.

При зсуві вліво (<<) звільнені розряди обнуляються. При зсуві вправо (>>) звільнені біти заповнюються нулями, якщо перший операнд беззнакового типу (тобто виконується логічний зсув), і знаковим розрядом - в іншому випадку (виконується арифметичний зсув). Операції зсуву ніколи не приводять до переповнення і втрати значущості. Стандартні операції зсуву визначені для типів int, uint, long і utong.

Операції відношення та перевірки на рівність

Операції відношення (<, <-,>,> -, -,! -) порівнюють перший операнд з другим. Операнди повинні бути арифметичного типу. Результат операції - логічного типу, дорівнює true або false.

Порозрядні логічні операції

Порозрядні логічні операції (&, |, Л) застосовуються до цілочисельних операндів і працюють з їх двійковими представленням. При виконанні операцій операнди зіставляються побітно (перший біт першого операнда з першим бітом другого, другий біт першого операнда з другим бітом другого і т. Д.). Стандартні операції визначені для типів int, uint, long і ulong.

При поразрядной кон'юнкції, або порозрядному І (операція позначається &), біт результату дорівнює 1 тільки тоді, коли відповідні біти обох операндів рівні 1. При поразрядной диз'юнкції, або порозрядному АБО (операція позначається |), біт результату дорівнює 1 тоді, коли відповідний біт хоча б одного з операндів дорівнює 1.

При порозрядному виключає АБО (операція позначається ~) біт результату дорівнює 1 тільки тоді коли відповідний біт тільки одного з операндів дорівнює 1.

Умовні логічні операції

Умовні логічні операції І (&&) і АБО (||) найчастіше використовуються з операндами логічного типу. Результатом логічної операції є true або false.

Результат операції логічне І має значення true, тільки якщо обидва операнда мають значення true. Результат операції логічне АБО має значення true, якщо хоча б один з операндів має значення true.

Умовний оператор

Умовний оператор (? :) - Тернарний, тобто має три операнда. Його формат: операнд_1? операнд_2: операнд З

Перший операнд - вираз, для якого існує неявне перетворення до логічного типу. Якщо результат обчислення першого операнда дорівнює true, то результатом умовної операції буде значення другого операнда, інакше - третього операнда. Обчислюється завжди або другий операнд, або третій.

Умовну операцію часто використовують замість умовного оператора if (він розглядається в наступному розділі) для скорочення тексту програми.

Приклад застосування умовної операції:

```
using System:
namespace ConsoleApplicationl {class Classl {Static void MainO {
int a = 11, b = 4; int max = b > a? b: a;
Console.WriteLine (max): // Результат 11 }
}
}
```

Операції присвоювання

Операції присвоювання (=, + =, -=, *= і т. д.) задають нове значення змінної. Формат операції простого присвоювання (=):

```
змінна = вираз
```

Механізм виконання операції присвоювання такий: обчислюється вираз і його результат заноситься в пам'ять за адресою, що визначається ім'ям змінної, що знаходиться зліва від знака операції. Те, що раніше зберігалося в цій області пам'яті, природно, втрачається. Схематично це корисно уявити собі так:

```
змінна <- вираз
```

Початківці часто роблять помилку, сприймаючи присвоювання як

аналог рівності в математиці. Щоб уникнути цієї помилки, треба розуміти механізм роботи оператора присвоювання.

Розглянемо для цього останній приклад (x = x + 0.5). Спочатку з комірки пам'яті, в якій зберігається значення змінної x, вибирається це значення. Потім до нього додається 0.5, після чого вийшов результат записується в ту ж саму осередок, а те, що зберігалося там раніше, втрачається безповоротно. Оператори такого виду застосовуються в програмуванні дуже широко.

Для правого операнда операції присвоювання повинно існувати неявне перетворення до типу лівого операнда. Наприклад, вираз цілого типу можна привласнити дійсної змінної, бо цілі числа є підмножиною речових, і інформація при такому присвоєнні не губиться:

Речова змінна - целий вираз;

У складних операціях присвоєння (+ =, -=, *= і т. д.) при обчисленні виразу, що стоїть в правій частині, використовується значення з лівої частини. Наприклад, при додаванні з привласненням до другого операнд додається перший, і результат записується в перший операнд, тобто вираз a + = 5 є більш компактною записом виразу a = a + 5.

Результатом операції складного присвоювання є значення, записане в лівий операнд.

Математичні функції - клас Math

У виразах часто використовуються математичні функції, наприклад синус або піднесення до степені. Вони реалізовані в класі Math, визначеному в просторі імен System. За допомогою методів цього класу можна обчислити:

- тригонометричні функції: Sin, Cos, Tan;
- обернені тригонометричні функції: ASin, ACos, ATan, ATan2;
- гіперболічні функції: Tanh, Sinh, Cosh;
- експоненту і логарифмічні функції: Exp, Log, Log 10;
- модуль (абсолютну величину), квадратний корінь, знак: Abs, Sqrt, Sign;

- округлення: Ceiling, Floor, Round;
- мінімум, максимум: Min, Max;
- степінь: Pow;
- залишок від ділення: DivRem.

Крім того, у класу є два корисних поля: число π і число e.

Опис методів і полів наведено в табл. 3.8.

Таблиця 3.8. Основні поля і статичні методи класу Math

Ім'я	Опис	Результат	Пояснення
Abs	Модуль	Перевантажений	| r | записується як Abs (x)
Acos	Арккосинус	double	Acos (double x)
Asin	Арксинус	double	Asin (double x)
Atan	Арктангенс	double	Atan (double x)
Atan2	Арктангенс	double	Atan2 (double x, double y) - кут, тангенс якого є результат ділення y на x
Ceiling	Округлення до більшого цілого	double	Ceiling (double x)
Cos	Косинус	double	Cos (double x)
Cosh	Гіперболічний косинус	double	Cosh (double x)
DivRem	залишок		DivRem (x, y, rem)
E	База натурального логарифма (число e)	double	2,71828182845905
Exp	Експонента	double	e^x записується як Exp (x)
Floor	Округлення до меншого цілого	double	Floor (double x)
IEEE Remainder	Залишок від ділення	double	IEEERemainder (double x, double y)
Log	Натуральний логарифм	double	Log_e x записується як Log(x)
Log10	Десятковий логарифм	double	$\log_{10}$ x записується як LogIO (x)
Max	Максимум з двох чисел	Перевантажений	Max (x, y)
Min	Мінімум з двох чисел	Перевантажений	Min (x, y)
PI	Значення числа я	double	3,14159265358979
Pow	Степінь	double	x^y записується як Pow (x, y)
Round	Округлення	Перевантажений	Round (3.1) дасть в результаті 3, Round (3.8) дасть в результаті 4
Sign	Знак числа	int	Аргументи перевантажені
Sin	Синус	double	Sin (double x)
Sinh	Гіперболічний синус	double	Sinh (double x)
Sqrt	Квадратний корінь	double	записується як Sqrt (x)
Tan	Тангенс	double	Tan (double x)
Tanh	Гіперболічний тангенс	double	Tanh (double x)

Лекція 4. Текстові рядки. Робота із рядковими даними. Базові маніпуляції рядками

Обробка текстової інформації є, ймовірно, однією з найпоширеніших завдань в сучасному програмуванні, і C# надає для її вирішення широкий набір засобів: окремі символи, масиви символів, змінювані і незмінні рядки і регулярні вирази.

Текстові дані в C# представляються за допомогою ключових слів string і char, які є скороченими позначеннями для типів System. String і System.Char. Як вже відомо, string представляє набір символів (наприклад, "Hello"), a char - одиночний символ в string (наприклад, 'H').

Символи

Символьний тип char призначений для зберігання символів в кодуванні Unicode. У цьому класі визначені статичні методи, що дозволяють задати вид і категорію символу, а також перетворити символ в верхній або нижній регістр і в число. Основні методи наведено в табл. 2.

Таблиця 2. Основні методи класу System.Char

Метод	Пояснення
GetNumericValue	Повертає числове значення символу, якщо він є цифрою, або -1 в іншому випадку
GetUnicodeCategory	Повертає категорію символа
IsControl	Повертає true, якщо символ є керуючим
IsDigit	Повертає true, якщо символ є десяткової цифрою
IsLetter	Повертає true, якщо символ є буквою
IsLetterOrDigit	Повертає true, якщо символ є буквою або цифрою
IsLower	Повертає true, якщо символ заданий у нижньому регістрі
IsNumber	Повертає true, якщо символ є числом (десятковим або шістнадцятковим)
IsPunctuation	Повертає true, якщо символ є знаком пунктуації
IsSeparator	Повертає true, якщо символ є роздільником
IsUpper	Повертає true, якщо символ записаний у верхньому регістрі
IsWhiteSpace	Повертає true, якщо символ є пробільним (пробіл, символ нового рядка та символ повернення каретки)
Parse	Перетворить рядок в символ (рядок повинен складатися з одного символу)
ToLower	Перетворює символ в нижній регістр
ToUpper	Перетворює символ у верхній регістр

```
static void CharFunctionality ()
{
Console.WriteLine ("=> char type Functionality:");
char myChar = 'a';
Console.WriteLine ("char.IsDigit ('a'): (0)", char.IsDigit (myChar));
Console.WriteLine ("char.IsLetter ('a'): (0)", char.IsLetter (myChar));
Console.WriteLine ("char.IsWhiteSpace ('Hello There', 5): {0}", char.IsWhiteSpace
("Hello There", 5));
Console.WriteLine ("char.IsWhiteSpace ('Hello There', 6): (0)", char.IsWhiteSpace
("Hello There", 6));
Console.WriteLine ("char.IsPunctuation ('?'): {0}", char.IsPunctuation ('?'));
Console.WriteLine ();
}
```

Для багатьох методів System.Char передбачені дві угоди щодо виклику: одиночний символ або рядок з числовим індексом, що вказує позицію символу, що перевіряється.

Для виведення коду символу в десятковому вигляді використовується явне перетворення до цілого типу. Явне перетворення із символів в рядки і назад в C# не існує, неявним же чином будь-який об'єкт, у тому числі і символ, може бути перетворено в рядок, наприклад.

Рядки типу string

Тип string, призначений для роботи з рядками символів в кодуванні Unicode. System.String містить методи для визначення довжини символьних даних, пошуку підрядку в поточному рядку і перетворення символів між верхнім і нижнім регістрами. **У табл. 3.5 перераховані деякі найцікавіші члени цього класу.**

Для рядків визначені такі оператори операції:

- присвоювання (=);
- перевірка на рівність (==);
- перевірка на нерівність (!=);
- звернення за індексом ([]);
- зчеплення (конкатенація) рядків (+).

Рядки рівні, якщо мають однакову кількість символів і збігаються посимвольно.

Звертатися до окремого елементу рядка за індексом можна тільки для отримання значення, але не для його зміни. Методи, що змінюють вміст рядка, насправді створюють нову копію рядка. Невикористані «старі» копії

автоматично видаляються збірником сміття.

Таблиця 3. Основні елементи класу System. String

Елемент	Тип елементу	Пяснення
Compare	Статичний метод	Порівняння двох рядків у лексикографічному (алфавітному) порядку. Різні реалізації методу дозволяють порівнювати рядки і підрядка з урахуванням і без урахування регістра і особливостей національного уявлення дат і т. Д.
CompareOrdInal	Статичний метод	Порівняння двох рядків за кодами символів. Різні реалізації методу дозволяють порівнювати рядки і підрядка
CompareTo	Метод	Порівняння поточного екземпляра рядки з іншого рядком
Concat	Статичний метод	Конкатенация строк. Метод допускає зчеплення довільного числа рядків
Copy	Статичний метод	Створення копії рядка
Empty	Статичне поле	Порожній рядок (тільки для читання)
Format	Статичний метод	Форматування відповідно до заданих специфікаторами формату (див. Далі)
IndexOf, IndexOfAny, LastIndexOf, LastIndexOfAny	Методи	Визначення індексів першого і останнього входження заданого підрядка або будь-якого символу із заданого набору
Insert	Метод	Вставка підрядка в задану позицію
Intern, IsInterned	Статичні методи	Повертає посилання на рядок, якщо така вже існує. Якщо рядки немає, Intern додає рядок у внутрішній пул, IsIntern повертає null
Join	Статичний метод	Злиття масиву рядків в єдину рядок. Між елементами масиву вставляються роздільники (див. Далі)
Length	Властивість	Довжина рядка (кількість символів)
PadLeft, PadRight	Методи	Вирівнювання рядки по лівому або правому краю шляхом вставки потрібного числа прогалин на початку або в кінці рядка
Remove	Метод	Видалення підрядка із заданої позиції
Replace	Метод	Заміна всіх входжень заданого підрядка або символу новими підрядком або символом
Split	Метод	Розділяє рядок на елементи, використовуючи задані роздільники. Результати поміщаються в масив рядків
StartsWith, EndsWith	Методи	Повертає true або false залежно від того, починається або закінчується рядок заданої підрядком
Substring	Метод	Виділення підрядка, починаючи з заданої позиції
ToCharArray	Метод	Перетворення рядка в масив символів
ToLower, ToUpper	Методи	Перетворення символів рядка до нижнього або верхнього регістру
Trim, TrimStart, TrimEnd	Методи	Видалення пробілів на початку і наприкінці рядка або тільки з одного її кінця (зворотні по відношенню до методів PadLeft і PadRight дії)

```
static void BasicStringFunctionality ()
{
Console.WriteLine ("=> Basic String functionality:"); string firstName = "Freddy";
// Значення firstName.
Console.WriteLine ("Value of firstName: (0)", firstName);
// Довжина firstname.
Console.WriteLine ("firstName has (0) characters.", FirstName.Length);
// FirstName у верхньому регістрі.
Console.WriteLine ("firstName in uppercase: {0}", firstName.ToUpper ());
// FirstName в нижньому регістрі.
Console.WriteLine ("firstName in lowercase: {0}", firstName.ToLower ());
// Чи містить firstName букву y?
Console.WriteLine ("firstName contains the letter y ?: {0}", firstName.Contains
("y"));
// FirstName після заміни.
Console.WriteLine ("firstName after replace: {0}", firstName.Replace ("dy", ""));
Console.WriteLine ();
}
```

Тут пояснювати особливо нічого: метод просто викликає різні члени, такі як ToUpper () і Contains (), на локальній змінній string для отримання різноманітних форматів і виконання перетворень. Нижче показаний результат:

```
***** Pun with Strings *****
=> Basic String functionality:
Value of firstName: Freddy firstName
has 6 characters. firstName in
uppercase: FREDDY '
firstName in lowercase: freddy firstName contains the letter
y ?: True firstName after replace: Fred
```

Конкатенація строк

Змінні string можуть бути зчеплені разом для створення рядків більшого розміру за допомогою операції + мови C#. Цей прийом формально називається конкатенацією рядків. Розглянемо наступну допоміжну функцію:

```
static void StringConcatenation ()
{
Console.WriteLine ("=> String concatenation:");
string si = "Programming the string s2 =" PsychoDrill (PTP) ";
string s3 = si + s2;
Console.WriteLine (s3);
Console.WriteLine ();
}
```

В результаті обробки компілятором символу + в C # видається виклик статичного методу Stmg.ConcatO. Тому конкатенацію рядків можна також здійснювати, викликаючи метод Strïng.ConcatO безпосередньо (хоча насправді це не дає якихось переваг, а лише збільшує обсяг введення).

Керуючі послідовності

Рядкові літерали C# можуть містити різні керуючі послідовності, які дозволяють уточнювати то, як символьні дані повинні виводитися на екран. Кожна керуюча послідовність починається з символу зворотної косої межі, за яким слідує знак, що інтерпретується. У табл. перераховані часто використовувані керуючі послідовності.

Таблиця – Керуючі послідовності в строкових літералах

Керуюча послідовність	Опис
\’	Вставляє в строковий літерал символ одинарної лапки
\"	Вставляє в строковий літерал символ подвійної лапки
\\	Вставляє в строковий літерал символ зворотної косої риси. Особливо корисна при визначенні шляхів до файлів і мережевих ресурсів
\a	Змушує систему видавати звуковий сигнал, який в консольних додатках може служити аудіо-підказкою користувачеві
\n	Вставляє символ нового рядка (на платформах Windows)
\r	Вставляє символ повернення каретки
\t	Вставляє в строковий літерал символ горизонтальної табуляції

Крім того, нижче наведений ще один приклад, в якому для залучення уваги кожен строковий літерал оснащений звуковим сигналом:

```
static void EscapeChars ()
{
Console.WriteLine ("=> Escape characters: \ a");
string strWithTabs = "Model \ tColor \ tSpeed \ tPet Name \ a
Console.WriteLine (strWithTabs);
Console.WriteLine ("Everyone loves V'Hello World \" \ a ");
Console.WriteLine ("C: \\ MyApp \\ bin \\ Debug \ a");
// Додати 4 порожніх рядки і знову видати звуковий сигнал.
Console.WriteLine ("All finished. \ N \ n \ n \ a");
Console.WriteLine ();
}
```

Визначення дослівних рядків

За рахунок додавання до строкових літералів префікса @ можна створювати так звані дослівні рядки. Дослівні рядки дозволяють відключати обробку керуючих послідовностей в літералах і виводити значення string в тому вигляді, в якому вони є. Ця можливість найбільш корисна при роботі з рядками, що представляють шляху до каталогів і мережевих ресурсів. Таким чином, замість використання керуючої послідовності \\ можна написати наступний код:

```
Console.WriteLine (@"C:\MyApp\bin\Debug");
```

Також зверніть увагу, що дослівні рядки можуть застосовуватися для зберігання пробілів у рядках, розділених на кілька рядків виводу.

Використовуючи дослівні рядки, можна також безпосередньо вставляти в літерали символи подвійної лапки, просто дублюючи лексему ":

```
Console.WriteLine (@ "Cerebus said" "Darrr! Pret-ty sun-sets");
```

Форматування рядків

Метод Format замінює всі входження параметрів у фігурних дужках значеннями відповідних змінних зі списку виводу. Після номера параметра можна задати мінімальну ширину поля виводу, а також вказати специфікатор формату, який визначає форму подання виведеного значення.

У загальному вигляді параметр задається наступним чином:

```
{n [, m [: специфікатор_формата [число]]}
```

Тут **n** - номер параметра. Параметри нумеруються з нуля, нульовий параметр замінюється значенням першої змінної зі списку виведення, перший параметр - другий змінної і т. д. Параметр **m** визначає мінімальну ширину поля, яке відводиться під виведене значення. Якщо виведеному числа досить меншої кількості позицій, невикористовувані позиції заповнюються пробілами. Якщо числу потрібно більше позицій, параметр ігнорується.

Специфікатор формату, як випливає з його назви, визначає формат виведення значення. Наприклад, специфікатор **C (Currency)** означає, що параметр повинен фрматуватися як валюта з урахуванням національних особливостей подання, а специфікатор **X (Hexadecimal)** задає шестнадцатеричну форму подання виведеного значення. Після деяких специфікаторів можна задати кількість позицій, що відводяться під дробову частину виведеного значення.

Можливо використовувати власні шаблони форматування. Після двокрапки задається вид виведеного значення посимвольно, причому на місці кожного символу може стояти або #, або 0. Якщо вказаний знак #, на цьому місці буде виведена цифра числа, якщо вона не дорівнює нулю. Якщо вказаний 0, буде виведена будь-яка цифра, в тому числі і 0.

Лекція 5. Оператор розгалуження if/else та оператор множинного вибору switch

Оператори розгалуження

Оператори розгалуження if і switch застосовуються для того щоб залежно від конкретних значень вихідних даних забезпечити виконання різних послідовностей операторів. Оператор if забезпечує передачу управління на одну з двох гілок обчислень, а оператор switch - на одну з довільного числа гілок.

Умовний оператор if

Умовний оператор if використовується для розгалуження процесу обчислень на два напрямки. Структурна схема оператора наведена на рис. 5.1.

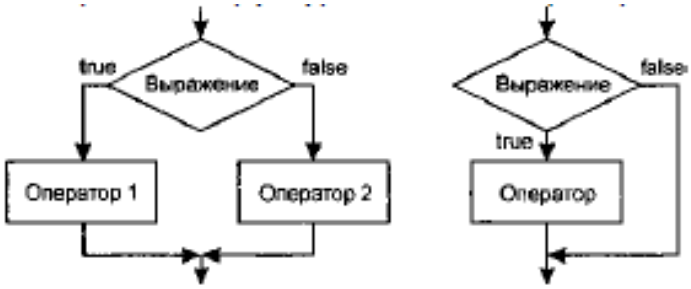

Рис. 5.1. Структурна схема умовного оператора

Формат оператора:

```
if (<логічний_вираз>) <оператор_1>; [Else <оператор_2>; ]
```

Спочатку обчислюється логічний вираз. Якщо він має значення true, виконується перший оператор, інакше - другий. Після цього управління передається на оператор, наступний за умовним. Гілка else може бути відсутня.

Якщо в гілці потрібно виконати декілька операторів, їх необхідно об'єднати в блок, інакше компілятор не зможе зрозуміти, де закінчується розгалуження. Блок може містити будь-які оператори, у тому числі опису та інші умовні оператори (але не може складатися з одних описів). Необхідно враховувати, що змінна, описана в блоці, поза блоком не існує.

Приклади умовних операторів:

У прикладі 1 відсутня гілка else. Подібна конструкція реалізує пропуск оператора, оскільки присвоювання або виконується, або пропускається

залежно від виконання умови.

Якщо потрібно перевірити кілька умов, їх об'єднують знаками логічних умовних операцій. Наприклад, вираз у прикладі 2 буде істинним в тому випадку, якщо виконається одночасно умова a <b і одна з умов в дужках. Якщо опустити внутрішні дужки, буде виконано спочатку логічне І, а потім - АБО.

Оператор в прикладі 3 обчислює найбільше значення з трьох змінних. Зверніть увагу на те, що компілятор відносить частину else до найближчого ключовому слову if.

Конструкції, подібні оператору в прикладі 4 (обчислюється найбільше значення з двох змінних), простіше і наочніше записувати у вигляді умовної операції, в даному випадку так:

```
max = b> a? b: a;
```

Поширена помилка початківців - невірний запис перевірки на приналежність діапазону. Наприклад, щоб перевірити умову 0 <x <1, не можна записати її в умовному операторі безпосередньо. Правильний спосіб запису: if (0 <x && x <1).

Слід уникати перевірки дійсних величин на рівність. Замість цього краще порівнювати модуль їх різниці з деяким малим числом. Це пов'язано з похибкою представлення дійсних значень у пам'яті:

```
float a, b; ...
if (a == b) ... // не рекомендується!
if (Math.Abs(a - b) <1e-6) ... // надійно!
```

Значення величини, з якою порівнюється модуль різниці, слід вибирати залежно від задачі, яка розв'язується. Знизу ця величина обмежена в класах Single і Double константою Epsilon (це мінімально можливе значення змінної таке, що 1.0 + Epsilon! = 1.0).

Оператор вибору switch

Оператор switch призначений для розгалуження процесу обчислень на кілька напрямів. Структурна схема оператора наведена на рис. 5.2.

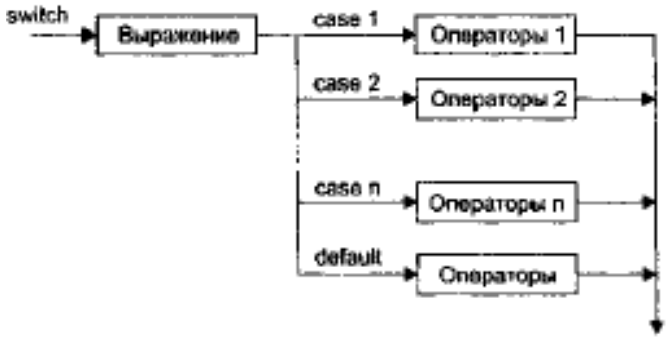

Рис. 5.2. Структурна схема оператора switch

Формат оператора:

```
switch (<вираз>) {
case <константний_вираз_1>: [<список_операторів_1>]
case <константний_вираз_2>: [<список_операторів_2>]
case <константний_вираз_3>: [<список_операторів_3>]
[default: <оператори>]
```

Виконання оператора починається з обчислення виразу. Тип виразу найчастіше цілочисельний або строковий. У загальному випадку вираз може бути будь-якого типу, для якого існує неявне перетворення до зазначених, а також типу переліку. Потім управління передається першому оператору зі списку, позначеного константним виразом, значення якого співпало з обчисленим.

Всі константні вирази повинні мати можливість неявного приведення до типу виразу в дужках. Якщо збігів не сталося, виконуються оператори, розташовані після слова default (а при його відсутності управління передається наступному за switch оператору).

Кожна гілка повинна закінчуватися явним оператором переходу, а саме оператором break, goto або return:

оператор break виконує вихід з самого внутрішнього з операторів switch;

оператор goto виконує перехід на вказану після нього мітку, зазвичай це мітка case однієї з нижчих гілок оператора switch;

оператор return виконує вихід з функції, в тілі якої він записаний.

Оператор goto зазвичай використовують для послідовного виконання кількох гілок перемикача, однак оскільки це порушує читабельність програми, такого рішення слід уникати.

Розглянемо приклад програми, що реалізує найпростіший калькулятор на чотири дії.

```
using System;
namespace ConsoleApplicationl
{
class Classl
{
static void Main()
{
string buf;
double a. b, res;
Console.WriteLine ("Введіть першу операнд:");
buf = Console.ReadLine();
a = double.Parse (buf);
Console.WriteLine ("Введіть знак операції");
char op = (char) Console.Read();
Console.ReadLine();
Console.WriteLine ("Введіть другий операнд:");
buf = Console.ReadLine();
b = double.Parse (buf);
bool ok = true;
switch (op)
{
case '+ ■; res = a + b: break:
case: res = a - b: break:
case: res = a * b; break:
case 7 *: res = a / b: break:
default: res = double.NaN; ok = false: break:
}
if (ok) Console.WriteLine ("Результат:" + res): else Console.WriteLine ("Неприпустима
операція");
}
}
}
```

Хоча наявність гілки default і не обов'язково, рекомендується завжди обробляти випадок, коли значення виразу не збігається ні з однією з констант. Це полегшує пошук помилок при налагодженні програми.

Лекція 6. Ітераційні конструкції. Цикл for.

Оператори циклу

Цикл — різновид керуючої конструкції у високорівневих мовах програмування, призначена для організації багаторазового виконання набору інструкцій. Цикл багаторазово повторює набір інструкцій, доки не будете виконана умова виходу з циклу. Одне повторення набору інструкцій (тіло циклу) також ще називають ітерацією. Приклади циклічних подій з навколишнього світу: зміна дня і ночі, зміна пір року, залізничний рух тощо.

Оператори циклу використовуються для обчислень, повторюваних багаторазово. У C # є чотири види циклів: цикл з передумовою **while**, цикл з післяумовою **do**, цикл з параметром **for** і цикл перебору **foreach**. Кожен з них складається з певної послідовності операторів.

Блок, заради виконання якого і організовується цикл, називається **тілом циклу**. Решта операторів служать для управління процесом повторення обчислень: це початкові установки, перевірка умови продовження циклу і модифікація параметра циклу (рис. 4.4).

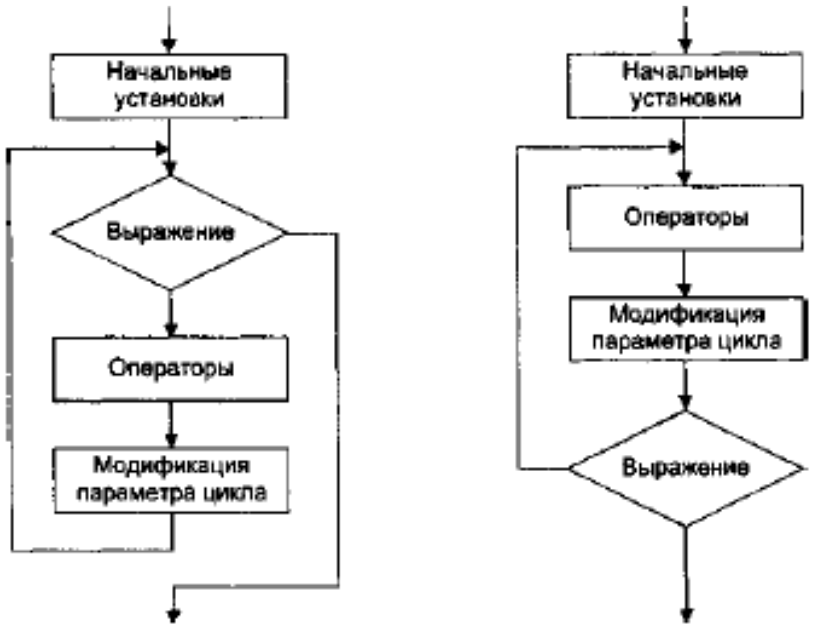

Рис. 4.4. Структурні схеми операторів циклу

Початкові установки служать для того, щоб до входу в цикл задати значення змінних, які в ньому використовуються.

Перевірка умови продовження циклу виконується на кожній ітерації

або до тіла циклу (тоді говорять про цикл із **передумовою**, схема якого показана на рис. 4.4, а), або після тіла циклу (цикл з **післяумовою**, рис. 4.4, б). Різниця між ними полягає в тому, що тіло циклу з післяумовою завжди виконується хоча б один раз, після чого перевіряється, чи треба його виконувати ще раз. Перевірка необхідності виконання циклу з передумовою робиться до тіла циклу, тому можливо, що він не виконається жодного разу.

Параметром циклу називається змінна, яка використовується при перевірці умови продовження циклу і примусово змінюється на кожній ітерації, причому, як правило, на одну і ту ж величину. Якщо параметр циклу цілочисельний, він називається **лічильником циклу**. Кількість повторень такого циклу можна визначити заздалегідь. Параметр є не у всякого циклу.

Цикл завершується, якщо умова його продовження не виконується. Можливо примусове завершення як поточної ітерації, так і циклу в цілому. Для цього служать оператори **break, continue, return і goto**. Передавати керування ззовні всередину циклу забороняється (при цьому виникає помилка компіляції).

Якщо в тілі циклу необхідно виконати більше одного оператора, необхідно укласти їх в блок за допомогою фігурних дужок.

Цикл з параметром for

Цикл з параметром має наступний формат:

for (ініціалізація; вираз; модифікації) оператор;

Ініціалізація служить для оголошення величин, що використовуються в циклі, і присвоєння їм початкових значень. У цій частині можна записати декілька операторів, розділених комою, наприклад:

```
for ( int i =0. j - 20; ,., int kt m:
for { k = 1. m = 0: ...
```

Областю дії змінних, оголошених в частині ініціалізації циклу, є цикл. Ініціалізація виконується один раз на початку виконання циклу.

Вираз типу **bool** визначає **умову виконання циклу**: якщо його результат дорівнює **true**, цикл виконується. Цикл з параметром реалізований як цикл з передумовою.

Модифікації виконуються після кожної ітерації циклу і служать зазвичай для зміни параметрів циклу. У частині модифікацій можна записати декілька операторів через кому, наприклад:

```
for ( int i = 0, j = 20: i < 5 && j > 10: 1++. j-- ) ...
```

Простий або складений оператор являє собою **тіло циклу**. Будь-яка з частин оператора for може бути опущена (але крапки з комою треба залишити на своїх місцях!).

Для прикладу обчислимо суму чисел від 1 до 100:

```
int s = 0;
for ( int 1=1; i <= 100; i++ ) s += i;
```

У лістингу 4.7 приведена програма, що виводить таблицю значень функції з лістингу 4.3.

Листинг 4.7. Таблица значений функции, полученных с использованием цикла for

```
using System;
namespace ConsoleApplication1
{   class Class1
    {   static void Main()
        {
            double Xn = -2, Xk = 12, dX = 2, t = 2, y;
            Console.WriteLine( "|   x    |    y   |";   // заголовок таблицы

            for ( double x = Xn; x <= Xk; x += dX )              // 1, 4, 5
            {
                y = t;                                           // 2
                if ( x >= 0 && x < 10 ) y = t * x;               // 2
```

Листинг 4.7 (*продолжение*)

```
                if ( x >= 10 )          y = 2 * t;               // 2
                Console.WriteLine( "| {0,6} | {1,6} |", x, y );   // 3
            }
        }
    }
}
```

Назвемо початкове значення аргументу Хп, кінцеве значення аргументу - Хк, крок зміни аргументу - dX і параметр t Всі величини дійсні. Програма повинна виводити таблицю, що складається з двох стовпців: значень аргументу і відповідних їм значень функції.

Опишемо алгоритм у словесній формі:

1. Взяти перше значення аргументу.
2. Визначити, якому з інтервалів воно належить, і обчислити значення функції за відповідною формулою.

3. Вивести рядок таблиці.

4. Перейти до наступного значення аргументу.

5. Якщо воно не перевищує кінцеве значення, повторити кроки 2-4, інакше закінчити.

Кроки 2-4 повторюються багаторазово, тому для їх виконання треба організувати цикл. Текст програми приведений в лістингу 4.3. Рядки програми позначені відповідними номерами кроків алгоритму.

Як бачите, в цьому варіанті програми все управління циклом зосереджено в його заголовку. Це робить програму зрозуміліше. Крім того, областю дії службової змінної x є цикл, а не вся функція, як це було в лістингу 4.3, що краще, оскільки змінна x поза циклом не потрібна.

У загальному випадку треба прагнути до мінімізації області дії змінних. Це полегшує пошук помилок в програмі.

Лекція 7. Ітераційні конструкції. Цикл foreach.

Як згадувалося в розділі 5, у мові C # визначений оператор циклу foreach, але його розгляд було відкладено до більш підходящого моменту. Тепер цей момент настав.

Оператор foreach служить для циклічного звернення до элементам *коллекции,* представляющей собой группу объектов. У C # визначено кілька видов коллекций, каждая из которых является массивом.Ниже приведена загальна форма оператора циклу foreach.

```
foreach (тип ім’я_змінної_цикла in колекція) оператор;
```

Тут `тип ім'я_змінної_циклу` позначає тип і ім'я змінної управління циклом, яка набуває значення наступного елемента колекції на кожному кроці виконання циклу foreach. А колекція позначає циклічно опитувану колекцію, яка тут і далі є масивом. Отже, тип змінної циклу повинен відповідати типу елемента масиву. Крім того, *тип* може позначатися ключовим словом var. У цьому випадку компілятор визначає тип змінної циклу, виходячи з типу елемента масиву. Це може виявитися корисним для роботи з певного роду запитами, як буде показано далі в даної книзі. Але, як правило, тип вказана явним чином.

Оператор циклу foreach діє наступним чином. Коли цикл починається, перший елемент масиву вибирається і присвоюється змінної циклу. На кожному наступному кроці ітерації вибирається наступний елемент масиву, який зберігається в змінної циклу. Цикл завершується, коли всі елементи масиву виявляться вибраними. Отже, оператор foreach циклічно опитує масив по окремих його елементів від початку і до кінця.

Слід, однак, мати в увазі, що змінна циклу в операторі foreach служить тільки для читання. Це означає, що, привласнюючи цієї змінної нове значення, не можна змінити вміст масиву.

Нижче наведено простий приклад застосування оператора циклу foreach. У цьому прикладі спочатку створюється цілочисельний масив і задається ряд його початкових значень, а потім ці значення виводяться, а по ходу справи обчислюється їх сума.

```
// Використовувати оператор циклу foreach
using System;
class ForeachDemo
{
static void Main ()
{
int sum = 0;
int [] nums = new int [10];
// Задати початкові значення елементів масиву nums
for (int i = 0; i <10; i ++)
nums [i] = i;
// Використовувати цикл foreach для виведення значень
// Елементів масиву та підрахунку їх суми
foreach (int x in nums)
{
Console.WriteLine ("Значення елемента одно:" + x);
sum + = x;
}
Console.WriteLine ("Сума дорівнює:" + sum);
}
}
```

Виконання наведеного вище коду дає наступний результат.

```
Значення елемента одно: 0
Значення елемента одно: 1
Значення елемента одно: 2
Значення елемента одно: 3
Значення елемента одно: 4
Значення елемента одно: 5
Значення елемента одно: 6
Значення елемента одно: 7
Значення елемента одно: 8
Значення елемента одно: 9
Сума дорівнює: 45
```

Як бачите, оператор foreach циклічно опитує масив по порядку індексування від самого першого до самого останнього його елемента.

Незважаючи на то що цикл foreach повторюється до тих пір, Бувай НЕ будуть опитані всі елементи масиву, його можна завершить преждевременно, воспользовавшись оператором break . Ниже приведен пример программы, в якої підсумовуються тільки п'ять перших елементів масиву nums.

```
// Використовувати оператор break для передчасного завершення циклу foreach. using
System;
class ForeachDemo
{
static void Main ()
{
int sum = 0;
int [] nums = new int [10];
// Задати початкові значення елементів масиву nums
for (int i = 0; i <10; i ++)
nums [i] = i;
```

```
// Використовувати цикл foreach для виведення значень
// Елементів масиву та підрахунку їх суми
foreach (int x in nums)
{
Console.WriteLine ("Значення елемента одно:" + x);
sum + = x;
if (x == 4) break; // Перервати цикл, як тільки індекс масиву досягне 4
}
Console.WriteLine ("Сума перших 5 елементів:" + sum);
}
}
```

Ось який результат дає виконання цієї програми.

```
Значення елемента одно: 0
Значення елемента одно: 1
Значення елемента одно: 2
Значення елемента одно: 3
Значення елемента одно: 4
Сума перших 5 елементів: 10
```

Абсолютно очевидно, що цикл foreach завершується після вибору і виведення значення п'ятого елементу масиву.

Оператор циклу foreach можна також використовувати для циклічного звернення до елементам багатовимірного масиву. У цьому випадку елементи багатовимірного масиву повертаються по порядку проходження рядків від перший до останньої, як демонструє наведений нижче приклад програми.

```
// Використовувати оператор циклу foreach для звернення до двовимірного масиву.
using System;
class ForeachDemo2
{
static void Main ()
{
int sum = 0;
int [,] nums = new int [3,5];
// Задати початкові значення елементів масиву nums.
for (int i = 0; i <3; i ++)
{
for (int j = 0; j <5; j ++)
nums [i, j] = (i + 1) * (j +1);
}
// Використовувати цикл foreach для виведення значень
// Елементів масиву та підрахунку їх суми.
foreach (int x in nums)
{
Console.WriteLine ("Значення елемента одно:" + x); sum + = x;
}
Console.WriteLine ("Сума дорівнює:" + sum);
}
}
```

Оператор foreach допускає циклічне звернення до масиву тільки в певному порядку: від початку і до кінця масиву, тому його застосування

здається, на перший погляд, обмеженим. Але насправді це не так. Велика кількість алгоритмів, найпоширенішим з яких є алгоритм пошуку, потрібен саме такий механізм. У якості прикладу нижче приведена програма, в якої цикл foreach використовується для пошуку в масиві певного значення. Як тільки це значення буде знайдено, цикл перерветься.

```
// Пошук в масиві за допомогою оператора циклу foreach.
Using System;
class Search
{
static void Main ()
{
int [] nums = new int [10];
int val;
bool found = false;
// Задати початкові значення елементів масиву nums.
for (int i = 0; i <10; i ++)
nums [i] = i; val = 5;
// Використовувати цикл foreach для пошуку заданого
// Значення в масиві nums.
foreach (int x in nums)
{
if (x == val)
{
found = true;
break;
}
}
}
if (found)
Console.WriteLine ("Значення знайдено!");
}
}
```

При виконанні цієї програми виходить наступний результат.

```
Значення знайдено!
```

Оператор циклу foreach відмінно підходить для такого застосування, оскільки при пошуку в масиві доводиться аналізувати кожен його елемент. До інших прикладів застосування оператора циклу foreach належить обчислення середнього, пошук мінімального чи максимального значення серед низки заданих значень, виявлення дублікатів тощо. Як буде показано далі в цієї книзі, оператор циклу foreach виявляється особливо корисним для роботи з різними типами колекцій.

Лекція 8. Ітераційні конструкції. Цикли while і do / while.

Формат оператора простий:

```
while (вираз) оператор;
```

Вираз повинен бути логічного типу. Наприклад, це може бути операція відношення або просто логічна змінна. Якщо результат обчислення виразу дорівнює true, виконується простий або складений оператор (блок). Ці дії повторюються до того моменту, поки результатом виразу не стане значення false. Після закінчення циклу управління передається на наступний за ним оператор.

Вираз обчислюється перед кожною ітерацією циклу. Якщо при першій перевірці вираз дорівнює false, цикл не виконається жодного разу.

В якості прикладу розглянемо програму, що виводить для аргументу x, що змінюється в заданих межах з заданим кроком, таблицю значень наступної функції:

$$y = \begin{cases} t, & x < 0 \\ tx, 0 & x < 10 \\ 2t, & x \geq 10 \end{cases}$$

Зверніть увагу на те, що умова продовження циклу записано в його заголовку і перевіряється до входу в цикл. Таким чином, якщо задати кінцеве значення аргументу, меншу початкового, навіть при негативному кроці цикл не буде виконаний жодного разу.

Параметром цього циклу, тобто змінної, управляє його виконанням, є x. Блок модифікації параметра циклу представлений оператором, що виконуються на кроці 4. Для перехода до наступного значення аргументу поточне значення нарощується на величину кроку і заноситься в ту ж змінну. Початківці часто забувають про модифікацію параметра, в результаті програма «зациклюється». Якщо з вами сталася така неприємність, спробуйте для завершення програми натиснути клавіші Ctrl + Break, а надалі перед запуском програми перевіряйте:

- присвоєно чи параметру циклу вірне початкове значення;

- чи змінюється параметр циклу на кожній ітерації циклу;
- чи вірно записано умова продовження циклу.

Поширеним прийомом програмування є організація нескінченного циклу із заголовком while (true) і примусовим виходом з тіла циклу з виконання будь-якої умови за допомогою операторів передачі управління.

Цикл з післяумовою do

Цикл з післяумовою має вигляд

```
do оператор while вираз;
```

Спочатку виконується простий або складений оператор, який утворює тіло циклу, а потім обчислюється вираз (воно повинно мати тип bool). Якщо вираз істинний, тіло циклу виконується ще раз і перевірка повторюється. Цикл завершується, коли вираз стане рівним false або в тілі циклу буде виконаний який- небудь оператор передачі управління.

Цей вид циклу застосовується в тих випадках, коли тіло циклу необхідно обов'язково виконати хоча б один раз, наприклад, якщо в циклі вводяться дані і виконується їх перевірка. Якщо ж такої необхідності немає, переважно користуватися циклом з передумовою.

Приклад програми, що виконує перевірку введення, приведений в лістингу:

```
using System.
namespace ConsoleApplication1
{   class Class1
    {   static void Main()
        {
            char answer;
            do
            {
                Console.WriteLine( "Купи слоника, а?" );
                answer = (char) Console.Read();
                Console.ReadLine();
            } while ( answer != 'y' );
```

Розглянемо ще один приклад застосування циклу з післяумовою - програму, що визначає корінь рівняння cos (x) = x методом розподілу навпіл з

точністю 0,0001.

Вихідні дані для цього завдання - точність, результат - число, що представляє собою корінь рівняння 1. Обидва значення мають дійсний тип.

Суть методу розподілу навпіл дуже проста. Здається інтервал, в якому є рівно один корінь (отже, на кінцях цього інтервалу функція має значення різних знаків). Обчислюється значення функції в середині цього інтервалу. Якщо воно того ж знака, що і значення на лівому кінці інтервалу, значить, корінь знаходиться в правій половині інтервалу, інакше - в лівій. Процес повторюється для знайденої половини інтервалу до тих пір, поки його довжина не стане менше заданої точності.

У наведеній далі програмою вихідний інтервал заданий за допомогою констант, значення яких взяті з графіка функції. Для рівнянь, що мають кілька коренів, можна написати додаткову програму, що визначає (шляхом обчислення та аналізу таблиці значень функції) інтервали, що містять рівно один корінь 2.

```
using System;
namespace ConsoleApplicationl
{
class Classl
{
static void Main()
{
double x, left = 0. right = 1;
do
{
x = C left + right )/ 2:
if ( ( Math.Cos(x) - x ) * ( Math.Cos(left) - left ) < 0 )
right = x;
else
left = x:
} while ( Math.Abst right - left ) < le-4 );
Console.WriteLine( "Корень равен " + x );
}
}
1
```

У цю програму для надійності дуже корисно додати підрахунок кількості виконаних ітерацій і примусовий вихід з циклу при перевищенні їх розумної кількості.

Будь-який цикл while може бути приведений до еквівалентного йому циклу for і навпаки.

Рекомендації по вибору оператора циклу

Оператори циклу взаємозамінні, але можна навести деякі рекомендації з вибору найкращого в кожному конкретному випадку.

Оператор do while зазвичай використовують, коли цикл потрібно обов'язково виконати хоча б раз, наприклад, якщо в циклі виробляється введення даних.

Оператором while зручніше користуватися в тих випадках, коли або число ітерацій заздалегідь невідомо, яких очевидних параметрів циклу немає, або модифікацію параметрів зручніше записувати не в кінці тіла циклу.

Оператор foreach застосовують для перегляду елементів різних колекцій об'єктів.

Оператор for підходить в більшості інших випадків. Однозначно - для організації циклів з лічильниками, тобто з цілочисельними змінними, які змінюють своє значення при кожному проході циклу регулярним чином (наприклад, збільшуються на 1).

Початківці часто роблять помилки при записі циклів. Щоб уникнути цих помилок, рекомендується:

- перевірити, чи всім змінним, яке трапляється в правій частині операторів присвоювання в тілі циклу, присвоєні до цього правильні початкові значення (а також можливо виконання інших операторів);
- перевірити, чи змінюється в циклі хоча б одна змінна, що входить до умова виходу з циклу;
- передбачити аварійний вихід з циклу після досягнення деякої кількості ітерацій (приклад наведено у наступному розділі);
- і, звичайно, не забувати про те, що якщо в тілі циклу потрібно виконати більше одного оператора, потрібно укладати їх у фігурні дужки. Оператори передачі управління

У C # є п'ять операторів, що змінюють природний порядок виконання обчислень:

- оператор безумовного переходу goto;

- оператор виходу з циклу break;
- оператор переходу до наступної ітерації циклу continue;
- оператор повернення з функції return;
- оператор генерації виключення throw.

Ці оператори можуть передати управління в межах блоку, в якому вони використані, і за його межі. Передавати керування всередину іншого блоку забороняється.

Оператор goto

Оператор безумовного переходу goto використовується в одній з трьох форм:

```
goto метка;
goto case константний_выраз;
goto default;
```

У тілі тієї ж функції повинна присутні рівно одна конструкція виду мітка: оператор;

Оператор goto мітка передає управління на позначений оператор. Метка - це звичайний ідентифікатор, областю видимості якого є функція, в тілі якої він заданий. Мітка повинна знаходитися в тій же області видимості, що і оператор переходу. Використання цієї форми оператора безумовного переходу виправдано у двох випадках:

- примусовий вихід вниз по тексту програми з декількох вкладених

циклів або перемикачів;

- перехід з декількох точок функції вниз по тексту в одну точку

(наприклад, якщо перед виходом з функції необхідно завжди виконувати які-

небудь дії).

В інших випадках для запису будь-якого алгоритму існують більш відповідні конструкції, а використання оператора goto призводить тільки до ускладнення структури програми і утруднення налагодження. Застосування цього оператора порушує принципи структурного і модульного

програмування, по яким всі блоки, що утворюють програму, повинні мати тільки один вхід і один вихід.

Друга і третя форми оператора goto використовуються в тілі оператора вибору switch. Оператор goto case константний_вираз передає управління на відповідну вирази зі сталими гілка, а оператор goto default - на гілку default. Треба зазначити, що реалізація оператора вибору в C # на рідкість невдала, і наявність у ньому оператора безумовного переходу ускладнює розуміння програми, тому краще обходитися без нього.

Оператор break

Оператор break використовується всередині операторів циклу або вибору для переходу в точку програми, що знаходиться безпосередньо за оператором, всередині якого знаходиться оператор break.

Оператор continue

Оператор переходу до наступної ітерації поточного циклу continue пропускає всі оператори, що залишилися до кінця тіла циклу, і передає керування на початок наступної ітерації.

Лекція 9. Алгоритми з використанням вкладених циклів. Розробка програм та алгоритмів накопичення сум

Оператор циклу for відноситься до самих універсальних операторів мови C #, скільки він допускає різні варіанти свого застосування. Деякі різновиди оператора циклу for розглядаються нижче.

```
// Выяснить, является ли число простым. Если оно
// непростое, вывести наибольший его множитель.
using System;
class FindPrimes
{
static void Main()
{
int num;
int i;
int factor;
bool isprime;
for(num = 2; num < 20; num++)
{
isprime = true; factor =0;
// Выяснить, делится ли значение переменной num нацело.
for(i=2; i <= num/2; i++)
{
if((num % i) == 0)
{
// Значение переменной num делится нацело.
// Следовательно, это непростое число
isprime = false;
factor = i;
}
}
if (isprime)
Console.WriteLine(num + " – простое число.");
else
Console.WriteLine("Наибольший множитель числа " + num +" равен " + factor);
}
}
}
```

Ниже приведен результат выполнения этой программы,

```
2 – простое число
3 – простое число
Наибольший множитель числа 4 равен 2 5 – простое число
Наибольший множитель числа 6 равен 3 7 – простое число
```

Застосування декількох змінних управління циклом

Нижче наведено практичний приклад застосування декількох змінних управління циклом в операторі for. У цьому прикладі програми використовуються дві змінні управління одним циклом for для виявлення

найбільшого і найменшого множника цілого числа (в даному випадку - 100). Зверніть особливу увагу на умову закінчення циклу. Вона спирається на обидві змінні управління циклом.

```
// Использовать запятые в операторе цикла for для
// выявления наименьшего и наибольшего множителя числа.
using System;
class Comma
{
static void Main()
{
int i, j ;
int smallest, largest;
int num;
num = 100;
smallest = largest = 1;
for(i=2, j=num/2; (i <= num/2) & (j >= 2); i++, j--)
{
if((smallest == 1) & ((num % i) == 0))
smallest = i;
if ((largest == 1) & ((num % j) == 0 > )
largest = j;
}
Console.WriteLine("Наибольший множитель: " + largest);
Console.WriteLine("Наименьший множитель: " + smallest);
}
}
```

Завдяки застосуванню двох змінних управління циклом вдається виявити найменший і найбільший множники числа в одному циклі for. Зокрема, параметр I служить для виявлення найменшого множника. Спочатку його значення встановлюється рівним 2 і потім інкрементується до тих пір, поки не пре підвищить половину значення змінної num. А керуюча змінна j служить для виявлення найбільшого множника. Її значення спочатку встановлюється рівним половині значення змінної num і потім декрементируется до тих пір, поки не стане менше 2. Цикл продовжує виконуватися до тих пір, поки обидві змінні, і та j, не досягнуть своїх кінцевих значень. По завершенні циклу обидва множники виявляються виявленими.

Відсутні частини циклу

Ряд цікавих різновидів циклу for виходить в тому випадку, якщо залишити порожніми окремі частині визначення циклу. У C # допускається залишати порожніми будь або ж всі частини ініціалізації, умови і ітерації в операторі циклу for. В якості прикладу розглянемо таку програму.

У даному прикладі ітераційне вираз у визначенні циклу for виявляється

порожнім, тобто воно взагалі відсутнє. Замість цього змінна і, керуюча циклом, інкрементується в тілі самого циклу. Це означає, що всякий раз, коли цикл повторюється, значення змінної і перевіряється на рівність числу 10, але ніяких інших дій при цьому не відбувається. А оскільки змінна і инкрементується в тілі циклу, то сам цикл виконується звичайним чином, виводячи наведений нижче результат.

У даному прикладі змінна і инициализируется перед початком циклу, а не в самому циклі for. Як правило, змінна керування циклом ініціалізується в циклі for. Виведення ініціалюзуючої частини за межі циклу зазвичай робиться лише в тому випадку, якщо первісне значення даної змінної виходить в результаті складного процесу, який недоцільно вводити в оператор циклу for.

Безкінечний цикл

Якщо залишити порожнім вираз умови в операторі циклу for, то вийде нескінченний цикл, тобто такой цикл , який ніколи не закінчується.

Цикли без тіла

У C # допускається залишати порожнім тіло циклу for чи іншого циклу, по скільки порожній оператор з точки зору синтаксису цієї мови вважається дійсним. Цикли без тіла нерідко виявляються корисними. Наприклад, в наступній програмі цикл без тіла служить для отримання суми чисел від 1 до 5.

```
// Тело цикла может быть пустым.
using System;
class Empty3 {
static void Main() {
int i;
int sum =0;
// получить сумму чисел от 1 до 5 for(i = 1; i <= 5; sum += i++);
Console.WriteLine("Сумма равна " + sum);
}
}
```

Выполнение отой программы дает следующий результат.

```
Сумма равна 15
```

Зверніть увагу на те, що процес підсумовування виконується повністю в операторі циклу for, і для цього тіло циклу не потрібно. У цьому циклі особливу увагу звертає на себе ітераційний вираз.

Подібні оператори не повинні вас бентежити. Вони часто зустрічаються в програм мах, професійно написаних на C #, і стають цілком зрозумілими, якщо розібрати їх по частинах. Дослівно Наведений вище оператор означає наступне: скласти зі значенням змінної sum результат підсумовування значень змінних sum і i, а потім інкрементувати значення змінної i. Отже, даний оператор рівнозначний наступній послідовності операторів.

Застосування оператора break для виходу з циклу

За допомогою оператора break можна спеціально організувати негайний ви хід з циклу в обхід будь-якого коду, що залишився в тілі циклу, а також минаючи переврку умови циклу. Коли в тілі циклу зустрічається оператор break, цикл завершується, а програма відновлюється з оператора, наступного після цього циклу. Розглянемо простий приклад програми.

Як бачите, цикл for організований для виконання в межах від -10 до +10, але, не дивлячись на це, оператор break перериває його раніше, коли значення змінної i стає позитивним.

Оператор break можна застосовувати в будь-якому циклі, передбаченому в C #.

А тепер розглянемо практичний приклад застосування оператора break. У наведеній нижче програмі виявляється найменший множник числа.

```
class FindSmallestFactor
{
static void Main()
{
int factor = 1;
int num = -1000;
for (int i=2; i <= num/i; i++)
{
if(num == 0)
{
factor = i;
break; // прервать цикл, как только будет
// выявлен наименьший множитель числа
}
}
Console.WriteLine{"Наименьший множитель равен " + factor);
}
}
```

Результат виконання цієї програми виглядає наступним чином.

```
Наименьший множитель равен 2
```

Оператор break перериває виконання циклу for, як тільки буде виявлений найменший множник числа. Завдяки такому застосуванню оператора break виключає опробування будь-яких інших значень після виявлення найменшого множилися ля числа, а отже, і неефективне виконання коду.

Якщо оператор break застосовується в цілому ряді вкладених циклів, то він прерії кість виконання тільки самого внутрішнього циклу. В якості прикладу розглянемо наступну програму.

Як бачите, оператор break з внутрішнього циклу викликає переривання тільки цього циклу, а на виконання зовнішнього циклу він не робить ніякого впливу.

Відносно оператора break необхідно також мати на увазі наступне. По - перше, в тілі циклі може бути присутнім кілька операторів break, але використовувати їх слід дуже акуратно, оскільки надмірна кількість операторів break зазвичай призводить до порушення нормальної структури коду. І по-друге, оператор break, що виконує вихід з оператора switch, впливає тільки на цей оператор, але не на осяжні цикли.

Застосування оператора continue

За допомогою оператора continue можна організувати передчасне закінчення кроку ітерації циклу в обхід звичайної структури управління циклом. Оператор continue здійснює примусовий перехід до наступного кроку циклу, пропускає будь-який код, що залишився невиконаним. Таким чином, оператор continue служить свого роду доповненням оператора break. У наведеному нижче прикладі про грами оператор continue використовується як допоміжний засіб для виведення парних чисел в межах від 0 до 100.

```
// Применить оператор continue.
using System;
class ContDemo
{
static void Main()
{
// вывести четные числа от 0 до 100.
for (int i = 0; i <= 100; i++)
{
if((i % 2) != 0) continue; // перейти к следующему шагу итерации
Console.WriteLine(i);
}
```

```
}
}
```

У даному прикладі виводяться тільки парні числа, оскільки при виявленні непарного числа крок ітерації циклу завершується передчасно в обхід виклику ме тода WriteLine ().

У циклах while і do- while оператор continue викликає передачу управління не посередньо умовному висловом, після чого триває процес виконання циклу. А в циклі for спочатку обчислюється ітераційне вираз, потім умовний вираз, після чого цикл триває.

Оператор continue рідко знаходить вдале застосування, зокрема, тому, що в C # надається багатий набір операторів циклу, що задовольняють більшу частину прикладних потреб. Але в тих особливих випадках, коли потрібно передчасне переривання кроку ітерації циклу, оператор continue надає структурований спосіб здійснення такого переривання.

Для прикладу розглянемо програму обчислення значення функції ch x (гіперболічний косинус) з точністю e = 10^{-6} за допомогою нескінченного ряду Тейлора за формулою

$$y = 1 + \frac{x^2}{2!} + \frac{x^4}{4!} + \frac{x^6}{6!} + \ldots + \frac{x^{2n}}{2n!} + \ldots$$

Цей ряд сходиться при $|x|<\infty$. Для досягнення заданої точності потрібно підсумовувати члени ряду, абсолютна величина яких більше e. Для ряду, що сходиться, модуль члена рядау C_n при збільшенні n прагне до нуля. При деякому n нерівність $|C_n|\geq\varepsilon$ перестає виконуватися і обчислення припиняються.

Алгоритм рішення задачі виглядає так: задати початкове значення суми ряду, а потім багаторазово обчислювати черговий член ряду і додавати його до раніше знайденої сумі. Обчислення закінчуються, коли абсолютна величина чергового члена ряду стане менше заданої точності.

До виконання програми передбачити, скільки членів ряду потрібно підсумувати, неможливо. У циклі такого роду є небезпека, що він ніколи не завершиться - як через можливість помилок в обчисленнях, так і через

обмежену область збіжності ряду (даний ряд сходиться на всій числовій осі, але існують ряди Тейлора, які сходяться тільки для певного інтервалу значень аргументу). Тому для надійності програми необхідно передбачити аварійний вихід з циклу з печаткою попереджуючого повідомлення по досягненні деякого максимально допустимої кількості ітерацій. Для виходу з циклу застосовується оператор break.

Пряме обчислення члена ряду за наведеною загальною формулою, коли x підноситься до ступеня, обчислюється факторіал, а потім чисельник ділиться на знаменник, має два недоліки, які роблять цей спосіб непридатним. Перший недолік - велика похибка обчислень. При піднесенні в ступінь і обчисленні факторіала можна отримати дуже великі числа, при розподілі яких один на одного відбудеться втрата точності, оскільки кількість значущих цифр, збережених в комірці пам'яті, обмежено 1. Другий недолік пов'язаний з ефективністю обчислень: як легко помітити, при обчисленні чергового члена ряду нам вже відомий попередній, тому обчислювати кожен член ряду «спочатку» нераціонально.

```
using System;
namespace ConsoleApplicationl
{
class Classl
{
static void MainO
{
double e = le-6;
const int Maxlter = 500;  // ограничитель количества итераций
Console.WriteLine( "Введите аргумент:" );
string buf = Console.ReadLine();
double x = Convert.ToDoubleC buf );
bool done = true;  // признак достижения точности
double ch = 1, y = ch;
for ( int n = 0; Math.Abs(ch) > e; n++ )
{
ch*=x*x/(2*n + l)/(2*n + 2);
y += ch;     // добавление члена ряда к сумме
if ( n > Maxlter )
{
done = false;
break; }
}
if ( done )
Console WriteLine( "Сумма ряда - " + y );
else
Console.WriteLine( "Ряд расходится" );
}
}
```

```
}
```

Отримання суми нескінченного ряду - приклад обчислень, які принципово неможливо виконати точно. У даному випадку ми задавали бажану похибка обчислень за допомогою значення e. Це значення не може бути менше, ніж найменше число, яке може бути представлене за допомогою змінної типу double, але при завданні такого значення точність результату фактично буде набагато нижче через похибок, що виникають при обчисленнях. Вони пов'язані з кінцівкою розрядної сітки.

У загальному випадку похибка результату складається з декількох частин:

- похибка постановки задачі (виникає при спрощення завдання);
- початкова похибка (точність представлення вихідних даних);
- похибка методу;
- похибки округлення і обчислення.

Специфіка машинних обчислень полягає в тому, що алгоритм, бездоганний з точки зору математики, при реалізації без урахування можливих похибок може призвести до отримання результатів, що не містять жодної вірної значущої цифри! Це відбувається, наприклад, при відніманні двох близьких значень або при роботі з дуже великими або дуже малими числами.

Лекція 10. Масиви. Ініціалізація масивів.

Масив представляє собою сукупність змінних одного типу з загальним для звернення до ним ім'ям. У C# масиви можуть бути як одновимірними, так і багатовимірними, хоча частіше всього застосовуються одномірні масиви. Масиви служать самим різним цілям, оскільки вони надають зручні засоби для об'єднання зв'язаних разом змінних. Наприклад, в масиві можна зберігати максимальні добові температури, зареєстровані в течія місяця, перелік біржових курсів або ж назви книг по програмуванню з домашньої бібліотеки.

Головне перевага масиву - в організації даних таким чином, щоб ними було простіше маніпулювати. Так, якщо мається масив, містить дивіденди, виплачувані по певної групі акцій, то, організувавши циклічне звернення до елементам цього масиву, можна без особливого праці розрахувати середній дохід від цих акцій. Крім того, масиви дозволяють організувати дані таким чином, щоб легко відсортувати їх.

Масивами в C# можна користуватися практично так само, як і в інших мовах програмування. Тим не менш у них мається один особливість: вони реалізовані в вигляді об'єктів. Саме тому їх розгляд було відкладено до тих пір, Бувай в цієї книзі не були представлені об'єкти. Реалізація масивів в вигляді об'єктів дає ряд істотних переваг, і далеко не самим останнім серед них є можливість утилізувати невикористовувані масиви засобам "збірки сміття".

Одномірні масиви

Одновимірний масив представляє собою список пов'язаних змінних. Такі списки часто застосовуються в програмуванні. Наприклад, в одновимірному масиві можна зберігати облікові номера активних користувачів мережі або поточні середні рівні досягнень бейсбольної команди.

Для того щоб скористатися масивом в програмою, потрібно двоетапна процедура, оскільки в C# масиви реалізовані в вигляді об'єктів. По перше, необхідно оголосити змінну, яка може звертатися до масиву. І по-друге, потрібно створити екземпляр масиву, використовуючи оператор new. Так, для

оголошення одновимірного масиву зазвичай застосовується наступна загальна форма:

```
тип [] імя_массіва = new тип [розмір];
```

де тип оголошує конкретний тип елемента масиву. Тип елемента визначає тип даних кожного елемента, що становить масив. Зверніть увага на квадратні дужки, які супроводжують тип. Вони вказують на то, що оголошується одновимірний масив. А розмір визначає число елементів масиву.

Звернемося до конкретному прикладу. У наведеної нижче рядку коду створюється масив типу int, Котрий складається з десяти елементів і зв'язується з змінної посилання на масив, іменованої sample.

```
int [] sample = new int [10];
```

У змінної sample зберігається посилання на область пам'яті, виділюваної для масиву оператором new. Ця область пам'яті повинна бути достатньо великий, щоб в ній могли зберігатися десять елементів масиву типу int.

Як і при створенні примірника класу, наведене вище оголошення масиву можна розділити на два окремих оператора. Наприклад:

```
int [] sample;
sample = new int [10];
```

У даному випадку змінна sample не посилається на якийсь певний фізичний об'єкт, коли вона створюється в перший операторі. І лише після виконання другого оператора ця змінна посилається на масив.

Доступ до окремому елементу масиву здійснюється по індексом: Індекс позначає становище елемента в масиві. У мовою C# індекс перший елемента всіх масивів виявляється нульовим. У Зокрема, масив sample складається з 10 елементів з індексами від 0 до 9. Для індексування масиву достатньо вказати номер необхідного елемента в квадратних дужках. Так, перший елемент масиву sample позначається як sample [0], а останній його елемент - як sample [9]. Нижче наведено приклад програми, в якій заповнюються всі 10 елементів масиву sample.

```
// Продемонструвати одновимірний масив.
using System;
class ArrayDemo
{
static void Main()
{
```

```
int [] sample = new int [10];
int i;
for (i = 0; i <10; i = i + 1)
sample [i] = i;
for (i = 0; i <10; i = i + 1)
Console.WriteLine ("sample [" + i + "]:" + sample [i]);
}
}
```

При виконанні цієї програми виходить наступний результат.

```
sample [0]: 0
sample [1]: 1
sample [2]: 2
sample [3]: 3
sample [4]: 4
sample [5]: 5
sample [6]: 6
sample [7]: 7
sample [8]: 8
sample [9]: 9
```

Масиви часто застосовуються в програмуванні тому, що вони дають можливість легко звертатися з великим числом взаємопов'язаних змінних. Наприклад, в наведеній нижче програмою виявляється середнє арифметичне ряду значень, що зберігаються в масиві nums, котрий циклічно опитується за допомогою оператора циклу for.

```
// Обчислити середнє арифметичне ряду значень.
using System;
class Average
{
static void Main()
{
int [] nums = new int [10];
int avg = 0;
nums [0] = 99;
nums [1] = 10;
nums [2] = 100;
nums [3] = 18;
nums [4] = 78;
nums [5] = 23;
nums [6] = 63;
nums [7] = 9;
nums [8] = 87;
nums [9] = 49;
for (int i = 0; i <10; i ++)
avg = avg + nums [i];
avg = avg / 10;
Console.WriteLine ("Середнє:" + avg);
}
}
```

Результат виконання цієї програми виглядає наступним чином.

```
Середнє: 53
```

Лекція 11. Організація доступу до елементів масиву. Заповнення значень елементів масивів

Присвоєння посилань на масиви

Присвоєння значення однією змінної посилання на масив другий змінної, по суті, означає, що обидві змінні посилаються на один і той ж масив, і в цьому відношенні масиви нічим не відрізняються від будь-яких інших об'єктів. Таке присвоювання не призводить ні до створенню копії масиву, ні до копіюванню вмісту одного масиву в інший. В якості прикладу розглянемо наступну програму.

```
// Присвоєння посилань на масиви.
using System;
class AssignARef
{
static void Main()
{
int i;
int [] nums1 = new int [10];
int [] nums2 = new int [10];
for (i = 0; i <10; i ++)
nums1 [i] = i;
for (i = 0; i <10; i ++)
nums2 [i] = -i;
Console.Write ("Вміст масиву nums1:");
for (i = 0; i <10; i ++)
Console.Write (nums1 [i] + "");
Console.WriteLine();
Console.Write ("Вміст масиву nums2:");
for (i = 0; i <10; i ++)
Console.Write (nums2 [i] + "");
Console.WriteLine ();
nums2 = nums1; // Тепер nums2 посилається на nums1
Console.Write ("Вміст масиву nums2 \ n" + "після присвоювання:");
for (i = 0; i <10; i ++)
Console.Write (nums2 [i] + "");
Console.WriteLine ();
// Далі оперувати масивом nums1 допомогою
// Змінної посилання на масив nums2.
nums2 [3] = 99;
Console.Write ("Вміст масиву numsl після зміни \ n" + "за допомогою змінної
nums2:");
for (i = 0; i <10; i ++) Console.Write (numsl [i] + "");
Console.WriteLine ();
}
}
```

Виконання цієї програми призводить до наступного результату.

```
Вміст масиву nums1: 0 1 2 3 4 5 6 7 8 9
Вміст масиву nums2: 0 -1 -2 -3 -4 -5 -6 -7 -8 -9
Вміст масиву nums2 після привласнення: 0 1 2 3 4 5 6 7 8 9
```

```
Вміст масиву nums1 після зміни
за допомогою змінної nums2: 0 1 2 99 4 5 6 7 8 9
```

Як бачите, після присвоювання змінної nums2 значення змінної nums1 обидві змінні посилання на масив посилаються на один і той же об'єкт.

Застосування властивості Length

Реалізація в C # масивів в вигляді об'єктів дає цілий ряд переваг. Одне з них полягає в тому, що з кожним масивом пов'язано властивість Length, містить число елементів, з яких може складатися масив. Отже, у кожного масиву мається спеціальне властивість, дозволяє визначити його довжину. Нижче наведено приклад програми, в якій демонструється цю властивість.

```
// Використовувати властивість Length масиву.
using System;
class LengthDemo
{
static void Main()
{
int [] nums = new int [10];
Console.WriteLine ("Довжина масиву nums дорівнює" + nums.Length);
// Використовувати властивість Length для ініціалізації масиву nums.
for (int i = 0; i <nums.Length; i ++)
nums [i] = i * i;
// А тепер скористатися властивістю Length
// Для виведення вмісту масиву nums.
Console.Write("Вміст масиву nums:");
for (int i = 0; i <nums.Length; i ++)
Console.Write (nums [i] + "");
Console.WriteLine ();
}
}
```

При виконанні цієї програми виходить наступний результат.

```
Довжина масиву nums дорівнює 10
Вміст масиву nums: 0 1 4 9 16 25 36 49 64 81
```

Зверніть увага на то, як в класі LengthDemo властивість nums.Length використовується в циклах for для управління числом повторюваних кроків циклу. У кожного масиву мається своя довжина, тому замість відстеження розміру масиву вручну можна використовувати інформацію про його довжині. Слід, однак, мати в увазі, що значення властивості Length ніяк не відображає число елементів, які в ньому використовуються на самому справі. Властивість Length містить лише число елементів, з яких може складатися масив.

Коли запитується довжина багатовимірного масиву, то повертається

загальне число елементів, з яких може складатися масив, як у наведеному нижче прикладі коду.

```
// Використовувати властивість Length тривимірного масиву
using System;
class LengthDemo3D
{
static void Main ()
{
int [,,] nums = new int [10, 5, 6];
Console.WriteLine ("Довжина масиву nums дорівнює" + nums.Length);
}
}
```

При виконанні цього коду виходить наступний результат.

```
Довжина масиву nums дорівнює 300
```

Як підтверджує наведений вище результат, властивість Length містить число елементів, з яких може складатися масив (в даному випадку - 300 (10 х 5 х 6) елементів). Тим НЕ менш властивість Length не можна використовувати для визначення довжини масиву в окремому його вимірі.

Завдяки наявності у масивів властивості Length операції з масивами під багатьох алгоритмах стають більш простими, а значить, і більш надійними. У якості прикладу властивість Length використовується в наведеної нижче програмою з метою поміняти місцями вміст елементів масиву, скопіювавши їх в зворотному порядку в інший масив.

```
// Поміняти місцями вміст елементів масиву.
using System;
class RevCopy
{
static void Main()
{
int i, j;
int [] nums1 = new int [10];
int [] nums2 = new int [10];
for (i = 0; i <nums1.Length; i ++)
nums1 [i] = i;
Console.Write ("Початковий вміст масиву:");
for (i = 0; i <nums2.Length; i ++)
Console.Write (nums1 [i] + "");
Console.WriteLine();
// Скопіювати елементи масиву numsl в масив nums2 в зворотному порядку.
if (nums2.Length> = numsl.Length) // перевірити, чи достатньо
// Довжини масиву nums2
for (i = 0, j = nums1.Length-1; i <nums1.Length; i ++, j--)
nums2 [j] = nums1 [i];
Console.Write ("Вміст масиву в зворотному порядку:");
for (i = 0; i <nums2.Length; i ++)
Console.Write (nums2 [i] + "");
Console.WriteLine();
}
```

```
}
```

Виконання цієї програми дає наступний результат.

```
Початковий вміст масиву: 0 1 2 3 4 5 6 7 8 9
Вміст масиву в зворотному порядку: 9 8 7 6 5 4 3 2 1 0
```

У даному прикладі властивість Length допомагає виконати два важливі функції. По-перше, воно дозволяє переконатися в тому, що довжини цільового масиву досить для зберігання вмісту вихідного масиву. І по-друге, воно надає умова для завершення циклу for, в якому виконується копіювання вихідного масиву в зворотному порядку. Звичайно, в цьому простому прикладі розміри масивів неважко з'ясувати і без властивості Length, але аналогічний підхід може бути застосований в цілому ряді інших, більш складних ситуацій.

Лекція 12. Принципи обробки даних в одномірних масивах. Методи сортування та пошуку даних.

У наведеної вище програмою початкові значення були задані для елементів масиву nums вручну в десяти окремих операторах присвоювання. Звичайно, така ініціалізація масиву абсолютно правильна, але то ж саме можна зробити набагато простіше. Адже масиви можуть ініціалізуватися, коли вони створюються. Нижче наведена загальна форма ініціалізації одновимірного масиву:

```
тип [] імя_массіва = {val1, val2, val3,..., valN};
```

де vall-valN позначають початкові значення, які присвоюються по черги, зліва направо і по порядку індексування. Для зберігання ініціалізаторів масиву в C# автоматично розподіляється достатній об'єм пам'яті. А необхідність користуватися оператором new явним чином відпадає сама собою. Як приклад нижче наведено покращений варіант програми, обчислює середнє арифметичне.

```
// Обчислити середнє арифметичне ряду значень.
using System;
class Average
{
static void Main()
{
int [] nums = {99, 10, 100, 18, 78, 23, 63, 9, 87, 49};
int avg = 0;
for (int i = 0; i <10; i ++)
avg = avg + nums [i];
avg = avg / 10;
Console.WriteLine ("Середнє:" + avg);
}
}
```

Цікаво, що при ініціалізації масиву можна також скористатися оператором new, хоча особливої потреби в цьому немає. Наприклад, наведений нижче фрагмент коду вважається вірним, але надлишковим для ініціалізації масиву nums в згаданої вище програмі.

```
int [] nums = new int [] {99, 10, 100, 18, 78, 23, 63, 9, 87, 49};
```

Незважаючи на свою надмірність, форма ініціалізації масиву з оператором new виявляється корисної в тому випадку, якщо новий масив присвоюється вже існуючої змінної посилання на масив. Наприклад:

```
int [] nums;
nums = new int [] {99, 10, 100, 18, 78, 23, 63, 9, 87, 49};
```

У даному випадку змінна nums оголошується в перший операторі і ініціалізується у другому.

І останнє зауваження: при ініціалізації масиву його розмір можна вказувати явним чином, але цей розмір повинен збігатися з числом ініціалізаторов. У Як приклад нижче наведено ще один спосіб ініціалізації масиву nums.

```
int [] nums = new int [10] {99, 10, 100, 18, 78, 23, 63, 9, 87, 49);
```

У цьому оголошенні розмір масиву nums задається рівним 10 явно.

Дотримання меж масиву

Кордони масиву в C # строго дотримуються. Якщо Кордони масиву НЕ досягаються або ж перевищуються, то виникає помилка при виконанні. Для того щоб переконатися в цьому, спробуйте виконати наведену нижче програму, в якої навмисно перевищуються межі масиву.

```
// Продемонструвати перевищення меж масиву.
using System;
class ArrayErr
{
static void Main()
{
int [] sample = new int [10];
int i;
// Відтворити перевищення меж масиву.
for (i = 0; i <100; i = i + 1)
sample [i] = i;
}
}
```

Як тільки значення змінної досягне 10, виникне виняткова ситуація типу IndexOutOfRangeException, пов'язана з виходом за межі індексування масиву, і програма передчасно завершиться.

Лекція 13. Багатовимірні масивами. Використання масивів у якості аргументів методу. Основні принципи використання класу System.Array

У програмуванні частіше всього застосовуються одномірні масиви, хоча і багатовимірні не так вже і рідкісні. Багатовимірним називається такий масив, Котрий відрізняється двома або більш вимірами, причому доступ до кожному елементу такого масиву здійснюється за допомогою певної комбінації двох або більше індексів.

Двовимірні масиви

Найпростішою формою багатовимірного масиву є двовимірний масив. Місцезнаходження будь-якого елемента в двовимірному масиві позначається двома індексами. Такий масив можна уявити в вигляді таблиці, на рядки якої вказує один індекс, а на стовпці - інший.

У наступної рядку коду оголошується двовимірний масив integer розмірами 10 x 20.

```
int [,] table = new int [10, 20];
```

Зверніть особливе увага на оголошення цього масиву. Як бачите, обидва його розміру розділяються комою. У першій частині цього оголошення синтаксичне позначення

```
[,]
```

означає, що створюється змінна посилання на двовимірний масив. Якщо ж пам'ять розподіляється для масиву з допомогою оператора new, то використовується наступне синтаксичне позначення.

```
int [10, 20]
```

У даному оголошенні створюється масив розмірами 10 x 20, але і в цьому випадку його розміри розділяються комою.

Для доступу до елементу двовимірного масиву слід вказати обидва індексу, розділивши їх коми. Наприклад, в наступної рядку коду елементу масиву table з координатами (3,5) присвоюється значення 10.

```
table [3, 5] = 10;
```

Нижче наведено більш наочний приклад в вигляді невеликої програми, в якої двовимірний масив спочатку заповнюється числами від 1 до 12, а потім виводиться його вміст.

```
// Продемонструвати двовимірний масив.
using System;
class TwoD
{
static void Main()
{
int t, i;
int [,] table = new int [3, 4];
for (t = 0; t <3; ++ t)
{
for (i = 0; i <4; ++ i)
{
table [t, i] = (t * 4) + i + 1;
Console.Write (table [t, i] + "");
}
Console.WriteLine();
}
}
}
```

У даному прикладі елемент масиву table [0,0] буде мати значення 1, елемент масиву table [0,1] - значення 2, елемент масиву table [0,2] - значення 3 і т.д. А значення елемента масиву table [2,3] виявиться рівним 12. На Рис. 7.1 показано схематично розташування елементів цього масиву і їх значень.

Масиви трьох і більше вимірів

УС# допускаються масиви трьох і більш вимірювань. Нижче наведена загальна форма оголошення багатовимірного масиву.

```
тип [, ...,] ім'я_масиву = new тип [розмір1, розмір2, ... розмірN];
```

Наприклад, в наведеному нижче оголошенні створюється тривимірний цілочисельний масив розмірами 4 x 10 x 3.

```
int [,,] multidim = new int [4, 10, 3];
```

А в наступному операторі елементу масиву multidim з координатами місцеположення (2,4,1) присвоюється значення 100.

```
multidim [2, 4, 1] = 100;
```

Нижче наведено приклад програми, в якої спочатку організовується тривимірний масив, містить матрицю значень 3 x 3 x 3, а потім значення елементів цього масиву підсумовуються по одній з діагоналей матриці.

```
// Підсумувати значення по одній з діагоналей матриці 3 x 3 x 3.
using System;
class ThreeDMatrix
{
static void Main()
{
int [,,] m = new int [3, 3, 3];
int sum = 0;
int n = 1;
for (int x = 0; x <3; x ++)
for (int y = 0; y < 3; y ++)
```

```
for (int z = 0; z <3; z ++)
m [x, y, z] = n ++;
sum = m [0, 0, 0] + m [1, 1,1] + m [2, 2, 2];
Console.WriteLine ("Сума значень по першій діагоналі:" + sum);
}
}
```

Ось який результат дає виконання цієї програми.

```
Сума значень по першій діагоналі: 42
```

Ініціалізація багатовимірних масивів

Д ля ініціалізації багатовимірного масиву достатньо укласти в фігурні дужки список ініціалізаторів кожного його розміру. Нижче в якості прикладу наведена загальна форма ініціалізації двовимірного масиву:

```
тип [,] імя_массіва = {
{val, val, val, ..., Val},
{val, val, val, ..., Val},
{val, val, val, ..., Val}
};
```

де val позначає инициализирующее значення, а кожен внутрішній блок - окремий ряд. Перше значення в кожному ряду зберігається на перший позиції в масиві, друге значення - на другий позиції і т.д. Зверніть увага на то, що блоки ініціалізаторів розділяються запитом, а після завершальній ці блоки закриває фігурної дужки ставиться крапка з комою.

У якості прикладу нижче наведена програма, в якої двовимірний масив sqrs ініціалізується числами від 1 до 10 і квадратами цих чисел.

```
// Ініціалізувати двовимірний масив. using System; class Squares {
static void Main () {int [,] sqrs = {
{1, 1},
{2, 4},
{3, 9},
{4, 16},
{5, 25},
{6, 36},
{7, 49},
{8, 64},
{9, 81},
{10, 100}
};
int i, j;
for (i = 0; i <10; i ++)
{
for (j = 0; j <2; j ++)
Console.Write (sqrs [i, j] + "");
Console.WriteLine ();
}
}
}
```

При виконанні цієї програми виходить наступний результат.

```
1 січня 4 лютого 3 вересня 16 квітня 25 травня
6    36
7    49
8    64
9    81 10100
```

Ступінчасті масиви

У наведених вище прикладах застосування двовимірного масиву, по суті, створювався так званий прямокутний масив. Двовимірний масив можна представити у вигляді таблиці, в якої довжина кожної рядки залишається незмінною по всьому масиву. Але в C # можна також створювати спеціальний тип двовимірного масиву, званий ступінчастим масивом. Ступінчастий масив представляє собою масив масивів, в якому довжина кожного масиву може бути різною. Отже, ступінчастий масив може бути використаний для складання таблиці з рядків різної довжини.

Ступінчасті масиви оголошуються з допомогою ряду квадратних дужок, в яких вказується їх розмірність. Наприклад, для оголошення двовимірного ступеневої масиву служить наступна загальна форма:

```
тип [] [] імя_массіва = new тип [розмір] [];
```

де розмір позначає число рядків в масиві. Пам'ять для самих рядків розподіляється індивідуально, і тому довжина рядків може бути різною. Наприклад, в наведеному нижче фрагменті коду оголошується ступінчастий масив jagged. Пам'ять спочатку розподіляється для його перший вимірювання автоматично, а потім для другий вимірювання вручну.

```
int [] [] jagged = new int [3] [];
jagged [0] = new int [4];
jagged [1] = new int [3];
jagged [2] = new int [5];
```

Тепер неважко зрозуміти, чому такі масиви називаються ступінчастими! Після створення ступеневої масиву доступ до його елементам здійснюється по індекс, що вказує в окремих квадратних дужках. Наприклад, в наступної рядку коду елементу масиву jagged, знаходиться на позиції з координатами (2,1), присвоюється значення 10.

```
jagged [2] [1] = 10;
```

Зверніть увага на синтаксичні відмінності в доступі до елементу ступеневої та прямокутного масиву.

У наведеному нижче прикладі програми демонструється створення двовимірного ступеневої масиву.

```
// Продемонструвати застосування східчастих масивів.
using System;
class Jagged
{
static void Main()
{
int [] [] jagged = new int [3] [];
jagged [0] = new int [4];
jagged [1] = new int [3];
jagged [2] = new int [5];
int i;
// Зберегти значення в першому масиві.
for (i = 0; i <4; i ++)
jagged [0] [i] = i;
// Зберегти значення в другому масиві.
for (i = 0; i <3; i ++)
jagged [1] [i] = i;
// Зберегти значення в третьому масиві.
for (i = 0; i <5; i ++)
jagged [2] [i] = i;
// Вивести значення з перших масиву.
for (i = 0; i <4; i ++)
Console.Write (jagged [0] [i] + "");
Console.WriteLine ();
// Вивести значення з другого масиву.
for (i = 0; i <3; i ++)
Console.Write (jagged [1] [i] + "");
Console.WriteLine ();
// Вивести значення із третього масиву.
for (i = 0; i <5; i ++)
Console.Write (jagged [2] [i] + "");
Console.WriteLine ();
}
}
```

Виконання цієї програми призводить до наступного результату.

```
0 1 2 3
0 1 2
0 1 2 3 4
```

Ступінчасті масиви знаходять корисне застосування не під всіх, а лише в деяких випадках. Так, якщо потрібно дуже довгий двовимірний масив, Котрий заповнюється не повністю, тобто такий масив, в якому використовуються не всі, а лише окремі його елементи, то для цієї мети ідеально підходить ступінчастий масив.

І останнє зауваження: ступінчасті масиви представляють собою масиви масивів, і тому вони не обов'язково повинні складатися з одновимірних масивів. Наприклад, у наведеній нижче рядку коду створюється масив двовимірних масивів.

```
int [] [,] jagged = new int [3] [,];
```

У наступної рядку коду елементу масиву jagged [0] присвоюється посилання на масив розмірами 4 x 2.

```
jagged [0] = new int [4, 2];
```

А в наведеної нижче рядку коду елементу масиву jagged [0] [1,0] присвоюється значення змінної i.

```
jagged [0] [1,0] = i;
```

Лекція 14. Методи (підпрограми). Методи як засіб структурної декомпозиції програм.

Метод - це функціональний елемент класу, який реалізує обчислення або інші дії, що виконуються класом чи примірником. Методи визначають поведінку класу.

Метод являє собою закінчений фрагмент коду, до якого можна звернутися по імені. Він описується один раз, а викликатися може стільки разів, скільки необхідно. Один і той же метод може обробляти різні дані, передані йому в якості аргументів.

Синтаксис методу:

```
[Атрибути] [специфікатори] тип ім’я_метода ([параметри])
тіло_метода
```

Розглянемо основні елементи опису методу. Перший рядок являє собою заголовок методу. Тіло методу, що задає дії, що виконуються методом, найчастіше являє собою блок - послідовність операторів у фігурних скобках1.

При описі методів можна використовувати специфікатори, що мають той же зміст, що й для полів, а також специфікатори virtual, sealed, override, abstract і extern, які будуть розглянуті в міру необхідності. Найчастіше для методів задається специфікатор доступу public, адже методи складають інтерфейс класу - те, з чим працює користувач, тому вони повинні бути доступні.

```
public double GetyO// метод для получения поля
{
return У;
}
```

Тип визначає, значення якого типу обчислюється за допомогою методу. Часто вживається термін «метод повертає значення», оскільки після виконання методу відбувається повернення в те місце викликає функції, звідки був викликаний метод, і передача туди значення виразу, записаного в операторі return (рис. 5.3). Якщо метод не повертає ніякого значення, в його заголовку задається тип void, а оператор return відсутня.

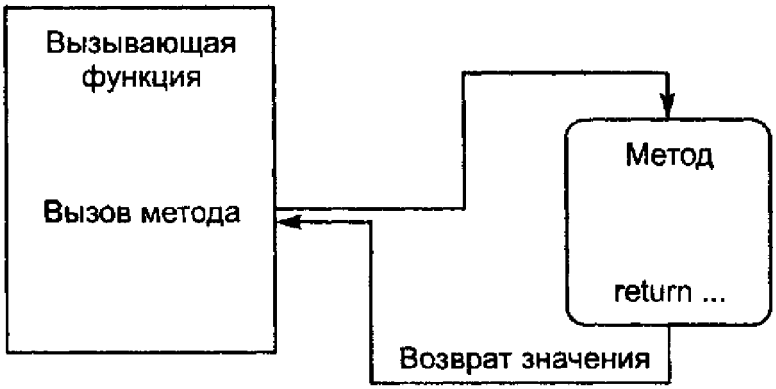

Рис. 5.3. Виклик методу

Параметри використовуються для обміну інформацією з методом. Параметр являє собою локальну змінну, яка при виклику методу приймає значення відповідного аргументу. Область дії параметра - весь метод.

Наприклад, щоб обчислити значення синуса для речовій величини x, ми передаємо її в якості аргументу в метод Sin класу Math, а щоб вивести значення цієї змінної на екран, ми передаємо її в метод WriteLine класу Console:

```
double x = 0.1; double y = Math.Sin(x);
Console.WriteLine(x);
```

При цьому метод Sin повертає в точку свого виклику речовинне значення синуса, яке присвоюється змінної y, а метод WriteLine нічого не повертає.

Параметри, описувані в заголовку методу, визначають безліч значень аргументів, які можна передавати в метод. Список аргументів при виклику ніби накладається на список параметрів, тому вони повинні попарно відповідати один одному. Правила відповідності докладно розглядаються в наступних розділах.

Для кожного параметра повинні задаватися його тип та ім'я. Наприклад, заголовок методу Sin виглядає наступним чином:

```
public static double Sin( double a );
```

Ім'я методу укупі з кількістю, типами і специфікаторами його параметрів являє собою сигнатуру методу - те, по чому один метод відрізняють від інших. У класі не повинно бути методів з однаковими сигнатурами.

У лістингу 5.2 в клас Demo додані методи установки і отримання значення поля y (насправді для подібних цілей використовуються не методи,

а властивості, які розглядаються трохи пізніше). Крім того, статичне поле s закрито, тобто визначене за умовчанням як private, а для його отримання описаний метод Gets, являло собою приклад статичного методу.

```
using System;
namespace ConsoleApplication1
{
    class Demo
    {
        public int a = 1;
        public const double c = 1.66;
        static string s = "Demo";
        double y;

        public double Gety()                // метод получения поля y
        {
            return y;
        }

        public void Sety( double y_ )       // метод установки поля y
        {
            y = y_;
        }

        public static string Gets()         // метод получения поля s
        {
            return s;
        }
    }

    class Class1
    {   static void Main()
        {
            Demo x = new Demo();
            x.Sety(0.12);                   // вызов метода установки поля y


           Console.WriteLine( x.Gety() );    // вызов метода получения поля y
           Console.WriteLine( Demo.Gets() ); // вызов метода получения поля s
//         Console.WriteLine( Gets() );     // при вызове из др. метода этого объекта
        }
    }
}
```

Як бачите, методи класу мають безпосередній доступ до його закритих полів. Метод, описаний зі специфікатором static, повинен звертатися тільки до статичних полів класу. Зверніть увагу на те, що статичний метод викликається через ім'я класу, а звичайний - через ім'я екземпляра.

Лекція 15. Параметри методів. Механізми передачі параметрів. Передача параметрів за значенням і за посиланням

Розглянемо більш детально, яким чином метод обмінюється інформацією з викликав його кодом. При виклику методу виконуються наступні дії:

1. Обчислюються виразів, що стоять на місці аргументів.
2. Виділяється пам'ять під параметри методу відповідно до їх типом.
3. Кожному з параметрів зіставляється відповідний аргумент (аргументи як би накладаються на параметри і заміщають їх).
4. Виконується тіло методу.
5. Якщо метод повертає значення, воно передається в точку виклику; якщо метод має тип void, управління передається на оператор, наступний після виклику.

При цьому перевіряється відповідність типів аргументів і параметрів і при необхідності виконується їх перетворення. При невідповідності типів видається діагностичне повідомлення.

```
using System;
namespace ConsoleApplication1
{   class Class1
    {
        static int Max(int a, int b)  // метод выбора максимального значения
        {
            if ( a > b ) return a;
            else         return b;
        }
        static void Main()
        {
            int a = 2, b = 4;
            int x = Max( a, b );                // вызов метода Max
            Console.WriteLine( x );             // результат: 4
```

```
            short t1 = 3, t2 = 4;
            int y = Max( t1, t2 );                  // вызов метода Max
            Console.WriteLine( y );                 // результат: 4

            int z = Max( a + t1, t1 / 2 * b );      // вызов метода Max
            Console.WriteLine( z );                 // результат: 5
        }
    }
}
```

У класі описаний метод Max, який вибирає найбільшу з двох переданих йому значень. Параметри описані як a і b. У методі Main виконуються три виклики Max. В результаті першого виклику методу Max передаються два аргументи того ж типу, що й параметри, у другому виклику - аргументи сумісного типу, в третьому - вираження.

Кількість аргументів повинна відповідати кількості параметрів. На імена ніяких обмежень не накладається: імена аргументів можуть як збігатися, так і не збігатися з іменами параметрів.

Існують два способи передачі параметрів: за значенням і за посиланням.

При передачі за значенням метод отримує копії значень аргументів, і оператори методу працюють з цими копіями. Доступу до початкових значень аргументів у методу немає, а отже, немає і можливості їх змінити.

При передачі по посиланню (за адресою) метод отримує копії адрес аргументів, він здійснює доступ до комірок пам'яті за цими адресами і може змінювати вихідні значення аргументів, модифікуючи параметри.

У C # для обміну даними між викликає і викликається функціями передбачено чотири типи параметрів:

□ параметри-значення;

□ параметри-посилання - описуються за допомогою ключового слова ref;

□ вихідні параметри - описуються за допомогою ключового слова out;

□ параметри-масиви - описуються за допомогою ключового слова params.

Ключове слово передує опису типу параметра. Якщо воно опущено, параметр вважається параметром-значенням. Параметр-масив може бути тільки один і повинен розташовуватися останнім у списку, наприклад:

```
public int Calculate( int a, ref int b, out int c, params int[] d ) ...
```

Параметри-значення

Параметр-значення описується в заголовку методу наступним чином: тип ім'я

Приклад заголовка методу, що має один параметр-значення цілого типу:

```
void P (int x)
```

Ім'я параметра може бути довільним. Параметр x являє собою локальну змінну, яка отримує своє значення з викликає функції при виклику методу. У метод передається копія значення аргументу.

Механізм передачі наступний: з комірки пам'яті, в якій зберігається змінна, передана в метод, береться її значення і копіюється в спеціальну область пам'яті - область параметрів. Метод працює з цією копією, отже, доступу до комірки, де зберігається сама змінна, не має. По завершенні роботи методу область параметрів звільняється. Таким чином, для параметрів-значень використовується, як ви здогадалися, передача за значенням. Ясно, що цей спосіб годиться тільки для величин, які не повинні змінитися після виконання методу, тобто для його вихідних даних.

При виклику методу на місці параметра, переданого за значенням, може перебувати вираз, а також, звичайно, його окремі випадки - змінна або константа. Має існувати неявне перетворення типу виразу до типу параметра1.

Наприклад, нехай в викликає функції описані змінні і їм до виклику методу привласнені значення:

```
int x - 1;
sbyte c = 1;
ushort y = 1;
```

Тоді наступні виклики методу P, заголовок якого був описаний раніше, будуть синтаксично правильними:

```
P( x );    P( c ),    P( y );    P( 200 );    P( x / 4 + 1 );
```

Параметри-посилання

У багатьох методах всі величини, які метод повинен отримати в якості

вихідних даних, описуються в списку параметрів, а величина, яку обчислює метод як результат своєї роботи, повертається в викликає код за допомогою оператора return. Очевидно, що якщо метод повинен повертати більше однієї величини, такий спосіб не годиться. Ще одна проблема виникає, якщо в методі потрібно змінити значення будь-яких переданих до нього величин. У цих випадках використовуються параметри-посилання.

Ознакою параметра-посилання є ключове слово ref перед описом параметра:

```
ref тип ім'я
```

Приклад заголовка методу, що має один параметр-посилання цілого типу: `void P (ref int x)`

При виклику методу в область параметрів копіюється не значення аргументу, а його адресу, і метод через нього має доступ до осередку, у якій зберігається аргумент. Таким чином, параметри-посилання передаються за адресою (частіше вживається термін «передача за посиланням»). Метод працює безпосередньо з змінної з викликає функції і, отже, може її змінити, тому якщо в методі потрібно змінити значення параметрів, вони повинні передаватися тільки по посиланню.

При виклику методу на місці параметра-посилання може знаходитися тільки посилання на ініціалізувала змінну точно того ж типу. Перед ім'ям параметра вказується ключове слово ref.

Вихідні дані передавати в метод за посиланням не рекомендується, щоб виключити можливість їх ненавмисного зміни.

Проілюструємо передачу параметрів-значень і параметрів-посилань на прикладі.

```
using System;
namespace ConsoleApplication1
{   class Class1
    {
        static void P( int a, ref int b )
        {
            a = 44; b = 33;
            Console.WriteLine( "внутри метода {0} {1}", a, b );
        }
        static void Main()
        {
            int a = 2, b = 4;
            Console.WriteLine( "до вызова     {0} {1}", a, b );
            P( a, ref b );
            Console.WriteLine( "после вызова  {0} {1}", a, b );
        }
```

Як бачите, значення змінної a в функції Main не змінилося, оскільки змінна передавалася за значенням, а значення змінної b змінилося тому, що вона була передана за посиланням.

Дещо інша картина вийде, якщо передавати в метод не величини значущих типів, а екземпляри класів, тобто величини посилальних типів. Як ви пам'ятаєте, змінна-об'єкт насправді зберігає посилання на дані, розташовані в динамічної пам'яті, і саме ця посилання передається в метод або за адресою, або за значенням. В обох випадках метод отримує в своє розпорядження фактична адреса даних і, отже, може їх змінити.

Різниця між передачею об'єктів за значенням і за посиланням полягає в тому, що в останньому випадку можна змінити саму посилання, тобто після виклику методу вона може вказувати на інший об'єкт.

Вихідні параметри

Досить часто виникає необхідність в методах, які формують кілька величин, наприклад, якщо в методі створюються об'єкти або ініціалізуються ресурси. У цьому випадку стає незручним обмеження параметрів-посилань: необхідність присвоювання значення аргументу до виклику методу. Це обмеження знімає специфікатор out. Параметри, що мають цей специфікатор, має бути обов'язково присвоєно значення всередині методу, компілятор за цим стежить. Зате в зухвалій коді можна обмежитися описом змінної без

ініціалізації.

Змінимо опис другого параметра, щоб він став вихідним.

```
using System;
namespace ConsoleApplication1
{   class Class1
    {
        static void P( int a, out int b )
        {
            a = 44; b = 33;
            Console.WriteLine( "внутри метода {0} {1}", a, b );
        }
        static void Main()
        {

           int a = 2, b;
           P( a, out b );
           Console.WriteLine( "после вызова  {0} {1}", a, b );
        }
    }
}
```

Лекція 16. Типи і аргументи методів. Рекурсивні методи

Рекурсивні функції — клас функцій, введений як уточнення класу обчислюваних функцій. В математиці загальноприйнятою є теза про те, що клас функцій, для обчислення яких існують алгоритми, при найширшому розумінні алгоритму, збігається з класом рекурсивних функцій. У зв'язку з цим, рекурсивні функції грають важливу роль в математиці та її застосуваннях, в першу чергу, в математичній логіці, основах математики та кібернетиці, як ефективно обчислювані функції. Тільки такі функції можна обчислювати на електронних обчислювальних машинах та інших цифрових пристроях.

При введенні класу ефективно обчислюваних функцій природнім чином виникає питання уточнення конструктивних об'єктів, на яких визначено ці функції. Клас всіх таких об'єктів широкий. В той же час, з допомогою методу арифметизації, запропонованого австрійським математиком Куртом Геделем, всі такі об'єкти легко зводяться до натуральних чисел. Тому рекурсивні функції було введено як функції, що визначені на множині натуральних чисел і набувають значення з тієї ж множини натуральних чисел. Перенесення понять і методів вироблених в теорії рекурсивних функцій на функції визначені на складніших конструктивних областях (множини слів деякого алфавіту, формул деякої теорії, графів тощо) не створює принципових ускладнень.

Будь-яка функція в програмі на мові С може викликатися рекурсивно, т. Е. Може викликати сама себе. Число рекурсивних викликів обмежена розміром стека. Відомості про параметри компоновщика, що визначають розмір стека, див. Розділ опису параметра компоновщика (/ STACK) Параметр / STACK (виділення пам'яті в стеку). При кожному виклику функції виділяється нова пам'ять для параметрів і змінних auto і register, щоб їх значення в попередніх (незакінчених) виклики не буде перезаписано. Параметри доступні безпосередньо тільки примірнику функції, в якому вони були створені. Наступним екземплярів функції попередні параметри

безпосередньо недоступні.

Зверніть увагу, що для змінних, оголошених з пам'яттю static, нова пам'ять при кожному рекурсивном виклик не потрібно. Їх пам'ять існує протягом часу життя програми. При кожному посиланні на таку змінну здійснюється доступ до тієї ж області пам'яті.

приклад

У наступному прикладі демонструються рекурсивні виклики:

```
static int Factorial(int x)
{
if (x == 0)
{
return 1;
}
else
{
return x * Factorial(x - 1);
}
}
```

Отже, тут у нас задається умова, що якщо вводиться число не дорівнює 0, то ми множимо дане число на результат цієї ж функції, в яку як параметр передається число x-1. Тобто відбувається рекурсивний спуск. І так, поки не дійдемо того моменту, коли значення варіанту не буде дорівнює одиниці.

При створенні рекурсивної функції в ній обов'язково повинен бути певний базовий варіант, який використовує оператор return і поміщається на початку функції. У випадку з факторіалом це if (x == 0) return 1

І, крім того, всі рекурсивні виклики повинні звертатися до підфункцій, які врешті-решт сходяться до базового варіанту. Так, при передачі в функцію позитивного числа при подальших рекурсивних викликах подфункций в них буде передаватися кожен раз число, менше на одиницю. І врешті-решт ми дійдемо до ситуації, коли число буде дорівнює 0, і буде використаний базовий варіант.

Іншим поширеним показовим прикладом рекурсивної функції служить функція, що обчислює числа Фіббоначчі. n-й член послідовності Фібоначчі визначається за формулою: f (n) = f (n-1) + f (n-2), причому f (0) = 0, а f (1) = 1.

```
static int Fibonachi(int n)
{
if (n == 0)
```

```
{
return 0;
}
if (n == 1)
{
return 1;
}
else
{
return Fibonachi(n - 1)   + Fibonachi(n - 2);
}
}
```

Лекція 17. Додаткові засоби мови програмування C#.

Клас є типом даних, визначеним користувачем. Він повинен представляти собою одну логічну сутність, наприклад, бути моделлю реального об'єкта або процесу. Елементами класу є дані і функції, призначені для їх обробки.

Опис класу містить ключове слово class, за яким слідує його ім'я, а далі у фігурних дужках - тіло класу, тобто список його елементів. Крім того, для класу можна задати його базові класи (предки) і ряд необов'язкових атрибутів і специфікаторів, визначають різні характеристики класу:

```
[ атрибутм ] [ специфікатори ] class им’якласа [ : предки ] тіло класа
```

Як бачите, обов'язковими є тільки ключове слово class, а також ім'я та тіло класу. Ім'я класу задається програмістом за загальними правилами C #. Тіло класу - це список описів його елементів, укладений у фігурні дужки. Список може бути порожнім, якщо клас не містить жодного елемента. Таким чином, найпростіше опис класу може виглядати так:

```
class Demo {}
```

Специфікатори визначають властивості класу, а також доступність класу для інших елементів програми. Можливі значення специфікаторів перераховані в табл. Клас можна описувати безпосередньо всередині простору імен або всередині іншого класу. В останньому випадку клас називається вкладеним. Залежно від місця опису класу деякі з цих специфікаторів можуть бути заборонені.

№	Специфікатор	Опис
1	**new**	Використовується для вкладених класів. Задає новий опис класу, натомість успадкованого від предка.Застосовується в ієрархіях об'єктів, розглядається в розділі 8 (див. С. 175)
2	**public**	Доступ не обмежений
3	**protected**	Використовується для вкладених класів. Доступ лише з елементів даного і похідних класів
4	**internal**	Доступ лише з даної програми (збірки) [1]
5	**protected internal**	Доступ лише з даного і похідних класів або з даної програми (збірки)

6	**private**	Використовується для вкладених класів. Доступ лише з елементів класу, всередині якого описаний даний клас
7	**abstract**	Абстрактний клас. Застосовується в ієрархіях об'єктів, розглядається в розділі 8 (див. С. 181)
8	**sealed**	Безплідний клас. Застосовується в ієрархіях об'єктів, розглядається в розділі 8 (див. С. 182)
9	**static**	Статичний клас. Введено в версію мови 2.0.Розглядається в розділі «Конструктори» (див. С.

Специфікатори 2-6 називаються специфікаторами доступу. Вони визначають, звідки можна безпосередньо звертатися до даного класу. Специфікатори доступу можуть бути присутніми в описі лише випадках, наведених у таблиці, а також можуть комбінуватися з іншими специфікаторами.

У цій главі ми будемо вивчати класи, які описуються в просторі імен безпосередньо (тобто не вкладені класи). Для таких класів допускаються тільки два специфікатора: public і internal. За замовчуванням, тобто якщо жоден специфікатор доступу не вказаний, мається на увазі специфікатор internal.

Клас є узагальненим поняттям, що визначає характеристики і поведінку деякого безлічі конкретних об'єктів цього класу, званих екземплярами, або об'єктами, класу.

Об'єкти створюються явним чи неявним чином, тобто або програмістом, або системою. Програміст створює екземпляр класу за допомогою операції new, наприклад:

```
Demo a = new DemoC);      // создание экземпляра класса Demo
Demo b = nevf DemoO;      // создание другого экземпляра класса Demo
```

Для кожного об'єкта при його створенні в пам'яті виділяється окрема область, в якій зберігаються його дані. Крім того, в класі можуть бути присутніми статичні елементи, які існують в єдиному екземплярі для всіх об'єктів класу. Часто статичні дані називають даними класу, а решта - даними екземпляра.

Функціональні елементи класу не тиражуються, тобто завжди зберігаються в єдиному екземплярі. Для роботи з даними класу

використовуються методи класу (статичні методи), для роботи з даними примірника - методи примірника, або просто методи.

Дотепер ми використовували в програмах тільки один вид функціональних елементів класу - методи. Поля і методи є основними елементами класу. Крім того, в класі можна задавати цілу гаму інших елементів: властивості, події, індексатори, операції, конструктори, деструктори, а також типи (рис. 5.1).

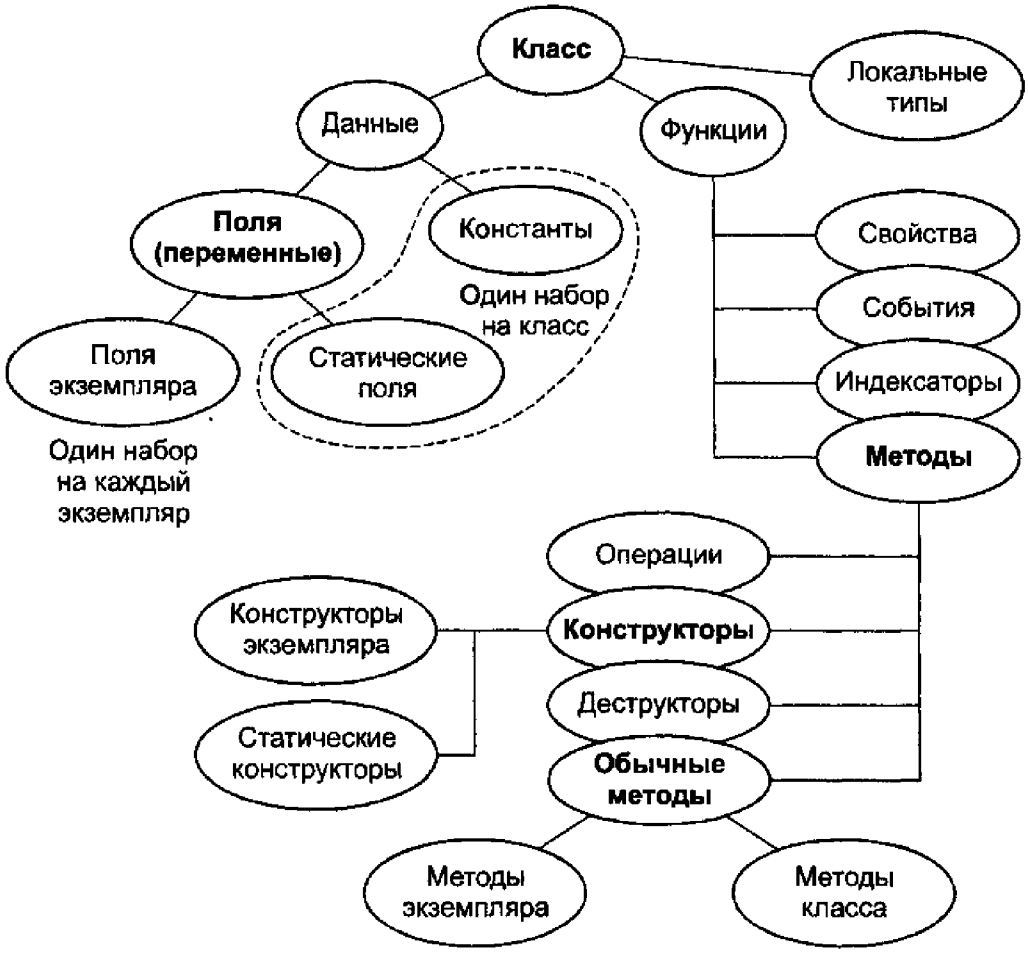

Рис. 5.1. Состав класса

Нижче наведено короткий опис всіх елементів класу (див. Також рис. 5.1):

□ Константи класу зберігають незмінні значення, пов'язані з класом.

□ Поля містять дані класу.

□ Методи реалізують обчислення або інші дії, що виконуються класом

чи примірником.

□ Властивості визначають характеристики класу в сукупності зі способами їх завдання та отримання, тобто методами запису і читання.

□ Конструктори реалізують дії з ініціалізації екземплярів або класу в цілому.

□ Деструктори визначають дії, які необхідно виконати до того, як об'єкт буде знищений.

□ Индексатором забезпечують можливість доступу до елементів класу по їх порядковому номеру.

□ Операції задають дії з об'єктами за допомогою знаків операцій.

□ Події визначають повідомлення, які може генерувати клас.

□ Типи - це типи даних, внутрішні стосовно до класу.

Перші п'ять видів елементів класу ми розглянемо в цьому розділі, а решта - в наступних. Але перш ніж почати вивчення, необхідно поговорити про присвоюванні і порівнянні об'єктів.

Присвоєння і порівняння об'єктів

Операція присвоювання розглядалася в розділі «Операції присвоювання». Механізм виконання присвоювання один і той же для величин будь- якого типу, як посилального, так і значимого, проте результати різняться. При присвоєнні значення копіюється значення, а при присвоєнні посилання - посилання, тому після присвоювання одного об'єкта іншому ми отримаємо два посилання, що вказують на одну і ту ж область пам'яті (рис. 5.2).

Малюнок ілюструє ситуацію, коли було створено три об'єкти, a, b і c, а потім виконано присвоювання b = c. Старе значення b стає недоступним і очищається складальником сміття. З цього випливає, що якщо змінити значення однієї величини посилального типу, це може відбитися на інший (в даному випадку, якщо змінити об'єкт через посилання c, об'єкт и також змінить своє значення).

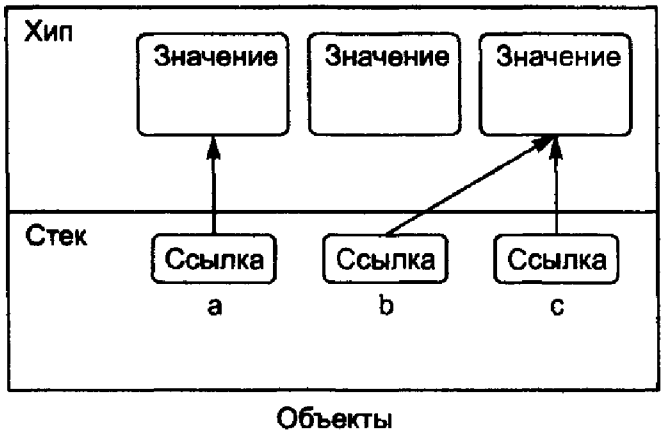

Рис. 5.2. Присвоєння об'єктів

Аналогічна ситуація з операцією перевірки на рівність. Величини значимого типу рівні, якщо рівні їх значення. Величини посилального типу рівні, якщо вони посилаються на одні й ті ж дані (на малюнку об'єкти b і c рівні, але a не одно b навіть при рівності їх значень або якщо вони обидві рівні null).