Изпитна тема № 2: **Обработка на колекции от данни**

*Сорс-контрол системи – дефиниция, видове (централизирани и децентрализирани) и команди за работа. Видове типове данни. Понятие за обект. Видове бройни системи, операции и преобразуване от една бройна система в друга. Изчисления в бройни системи. Едномерни и многомерни масиви – дефиниция, деклариране, описание и илюстрация на структурата, разработка на алгоритми върху масиви. Списъци – дефиниция, деклариране, описание на структурата, основни операции, разработка на алгоритми върху списъци, изводи за предимствата и недостатъците спрямо масивите. Дебъгване и работа с дебъгер – откриване на проблеми в програми с използването на дебъгер. Символни низове – дефиниция, деклариране, методи за работа, разработка на алгоритми за обработка на текст (извличане на подниз, замяна на низ и др.). Речници (хеш-таблици) – дефиниция, деклариране, устройство, методи за работа, решаване на задачи върху речници, разлика между ключ и стойност. Определяне на реда на изпълнение на фрагментите в кода (program flow). Създаване и/или поправка/допълване на вече съществуващи компютърни програми, решаващи изчислителни и алгоритмични задачи със средствата на програмен език.*

**Контрол на версиите (Version Control)-** Съдържа техники, практики и инструменти за работа със споделени файлове и програмен код, Има механизми за управление, контрол и проследяване на промените, Дефинира процеса на управление на промените, Описва какво се е случило в проекта с течение на времето, Разрешава конфликтите, възникнали при промените.

Git=разпределена система за контрол на програмния код (source-control system), Най-популярната в света (към момента), Свободна, софтуер с отворен код, Работи с локални и отдалечени хранилища, Git bash – команден интерфейс към Git, Върви на Linux, Mac OS и Windows (msysGit).

Основни Git команди- Клониране на съществуващо Git хранилище, Изтегляне и сливане на промени от отдалечено хранилище, Подготовка (добавяне / избор) на файлове за запис, Предаване (commit) към локалното хранилище, Проверка на статуса (промените) в локалното хранилище, Създаване на ново локално хранилище, Създаване на отдалечено, Изпращане на промени.

GitHub е #1 сайт в света за хостване на програмен код: Безплатен за проекти с отворен код, Платени планове за частни хранилища

GitHub предоставя: Git хранилище за програмен код, Тракер на задачи (бъг тракер), Табло на проекта (тип Канбан), Уики страници (документация)

**Типовете данни** представляват множества (диапазони) от стойности, които имат еднакви характеристики. Типовете данни се характеризират с:

- Име ;

- Размер (колко памет заемат);

- Стойност по подразбиране (default value).

Видове-Базовите типове данни в C# се разделят на следните видове:

- Целочислени типове – sbyte, byte, short, ushort, int, uint, long, ulong;

- Реални типове с плаваща запетая – float, double;

- Реални типове с десетична точност – decimal;

- Булев тип – bool;

- Символен тип – char;

- Символен низ (стринг) – string;

- Обектен тип – object.

|  |  |  |  |
| --- | --- | --- | --- |
| **Тип данни** | **Стойност по подразбиране** | **Минимална стойност** | **Максимална стойност** |
| **sbyte** | 0 | -128 | 127 |
| **byte** | 0 | 0 | 255 |
| **short** | 0 | -32768 | 32767 |
| **ushort** | 0 | 0 | 65535 |
| **int** | 0 | -2147483648 | 2147483647 |
| **uint** | 0u | 0 | 4294967295 |
| **long** | 0L | -9223372036854775808 | 9223372036854775807 |
| **ulong** | 0u | 0 | 18446744073709551615 |
| **float** | 0.0f | ±1.5×10-45 | ±3.4×1038 |
| **double** | 0.0d | ±5.0×10-324 | ±1.7×10308 |
| **decimal** | 0.0m | ±1.0×10-28 | ±7.9×1028 |
| **boolean** | false | Възможните стойности са две – **true** или **false** | |
| **char** | '\u0000' | '\u0000' | ‘\uffff’ |
| **object** | null | - | - |
| **string** | null | - | - |

Целочислените типове отразяват целите числа и биват sbyte, byte, short, ushort, int, uint, long и ulong.

Реалните типове в C# представляват реалните числа. Те се представят чрез плаваща запетая. Биват float и double се различават освен с порядъка на възможните стойности, които могат да заемат, и по точност.

Булевият тип се декларира с ключовата дума bool. Той има две стойности, които може да приема – true и false. Стойността по подразбиране е false. Използва се най-често за съхраняване на резултата от изчислява­нето на логически изрази.

Символният тип представя единичен символ. Той се декларира с ключо­вата дума char в езика C#. Минималната стойност, която може да заема типът char, е 0, а максималната – 65535. Стойностите от тип char представляват букви или други символи и се ограждат в апострофи.

Символните низове представляват поредица от символи. Декларират се с ключовата дума string в C#. Стойността им по подразбиране е null. Стринговете се ограждат в двойни кавички.

Обектният тип е специален тип, който се явява родител на всички други типове в .NET Framework. Декларира се с клю­човата дума оbject и може да приема стойности от всеки друг тип. Той представлява референтен тип, т.е. указател (адрес) към област от паметта, която съхранява неговата стойност.

**Бройните системи** (numeral systems) са начин за представяне (запис­ване) на числата, чрез краен набор от графични знаци наречени цифри. Към тях трябва да се добавят и правила за представяне на числата. Символите, които се използват при представянето на числата в дадена бройна система, могат да се възприемат като нейна азбука. Освен азбука, всяка бройна система има и основа. Основата е число, равно на броя различни цифри, използвани от системата за записване на числата в нея. Бройните системи се наричат позиционни (positional), тогава, когато мястото (пози­цията) на цифрите има значение за стойността на числото.

|  |  |  |  |
| --- | --- | --- | --- |
| **Двоична** | **Осмична** | **Десетична** | **Шестнадесетична** |
| 0000 | 0 | 0 | 0 |
| 0001 | 1 | 1 | 1 |
| 0010 | 2 | 2 | 2 |
| 0011 | 3 | 3 | 3 |
| 0100 | 4 | 4 | 4 |
| 0101 | 5 | 5 | 5 |
| 0110 | 6 | 6 | 6 |
| 0111 | 7 | 7 | 7 |
| 1000 | 10 | 8 | 8 |
| 1001 | 11 | 9 | 9 |
| 1010 | 12 | 10 | A |
| 1011 | 13 | 11 | B |
| 1100 | 14 | 12 | C |
| 1101 | 15 | 13 | D |
| 1110 | 16 | 14 | E |
| 1111 | 17 | 15 | F |

Освен позиционни, съществуват и непозиционни бройни системи, при които стойността на всяка цифра е постоянна и не зависи по никакъв начин от нейното място в числото.

Преминаване от двоична в десетична бройна система- всяка една двоична цифра се умножава по 2 на степен, позицията, на която се намира (в двоичното число). Накрая се извършва събиране на числата, получени за всяка от двоичните цифри, за да се получи десетичната равностойност на двоичното число.

Преминаване от десетична в двоична бройна система, се извършва преобразуване на десетичното число в двоично. За целите на преобразу­ването се извършва делене на две с остатък. Така се получават частно и остатък, който се отделя.

Преобразуване от двоични в шестнайсетични числа се извършва бързо и лесно, чрез разделяне на двоичното число на групи от по четири бита (разделяне на полубайтове). Ако броят на цифрите в числото не е кратен на четири, то се добавят водещи нули в старшите разряди. След разделянето и евентуалното добавяне на нули, се заместват всички полу­чени групи със съответстващите им цифри.

Преобразуване от шестнадесетична към десетична БС – Умножава се всяка цифра по основата на съответния степенен показател

Преобразуване на числа от десетична към шестнадесетична БС - Дели се на 16 и прибавяме остатъците в обратен ред

**Изчисления в бройни системи:**

Събиране в двоична бройна система - събирането става с помощта на следната таблица:

|  |  |  |
| --- | --- | --- |
| + | **0** | **1** |
| 0 | 0 | 1 |
| 1 | 1 | 10 |

Умножение в двоична бройна система - умножението става с помощта на следната таблица:

|  |  |  |
| --- | --- | --- |
| \* | **0** | **1** |
| 0 | 0 | 0 |
| 1 | 10 | 1 |

**Масивите** представля­ват съвкупности от променливи, които наричаме елементи. Елементите на масивите в C# са номерирани с числата 0, 1, 2, ... N-1. Тези номера на елементи се наричат индекси. Броят елементи в даден масив се нарича дължина на масива. Всички елементи на даден масив са от един и същи тип, независимо дали е примитивен или референтен.

Масивите могат да бъдат от различни размерности, като най-често използвани са едномерните и двумерните масиви. Едномерните масиви се наричат още вектори, а двумерните – матрици.

Масиви в C# декларираме по следния начин:

|  |
| --- |
| int[] myArray; |

В примера променливата myArray е името на масива, който е от тип (int[]), т.е. декларирали сме масив от цели числа. С [] се обозначава, че промен­ливата, която декларираме е масив от елементи, а не единичен елемент.

При декларация на променливата от тип масив, тя пред­ставлява референция (reference), която няма стойност (сочи към null), тъй като още не е заделена памет за елементите на масива.

В C# масив се създава с ключовата дума new, която служи за заделяне (алокиране) на памет:

|  |
| --- |
| int[] myArray = new int[6]; |

Преди да използваме елемент от даден масив той трябва да има начална стойност. В C# всички променливи, включително и елементите на масивите имат начална стойност по подразбиране (default initial value). Тази стойност е равна на 0 при числените типове или неин еквивалент при нечислени типове. Начални стойности можем да задавам и изрично. Това може да стане по различни начини. Ето един от тях:

|  |
| --- |
| int[] myArray = { 1, 2, 3, 4, 5, 6 }; |

Масивите по подразбиране са нулево-базирани, т.е. номерацията на елемен­тите започва от 0. Първият елемент има индекс 0, вторият 1 и т.н. Ако един масив има N елемента, то последният елемент се намира на индекс N-1.

Достъпът до елементите на масивите е пряк и се осъществява по индекс. Всеки елемент може да се достъпи с името на масива и съответния му индекс (пореден номер), поставен в квадратни скоби. Можем да осъще­ствим достъп до даден елемент както за четене така и за писане т.е. да го третираме като най-обикновена променлива.

Пример за достъп до елемент на масив:

|  |
| --- |
| myArray[index] = 100; |

**Многомерни масиви:** Всеки допустим в C# тип може да бъде използван за тип на елементите на масив. Масивите също може да се разглеждат като допустим тип. Така можем да имаме масив от масиви. Такива масиви ще наричаме двумерни, защото имат две измерения или още матрици (терминът идва от математиката). Масиви с повече от едно измерение ще наричаме многомерни.

Списък е линейна структура от данни, която съдържа поредица от елементи. Списъкът има свойството дължина (брой елементи) и елемен­тите му са наредени последователно. Списъкът позволява добавяне на елементи на всяко едно място, премахването им и последователното им обхождане.

Класът List<T>

При инициализацията на обект от тип List<T> указваме типа на елементите, който ще съдържа списъка, т.е. заместваме означения с T тип с някой истински тип данни (например число или символен низ).

Нека разгледаме случай, в който искаме да създадем списък от целочислени елементи. Можем да го направим по следния начин:

|  |
| --- |
| List<int> intList = new List<int>(); |

Създаденият по този начин списък може да съдържа като стойности само цели числа и не може да съдържа други обекти, например символни низове. Ако се опитаме да добавим към List<int> обект от тип string, ще получим грешка по време на компилация. List<T> съдържа списък от елементи (като масив, но разширяващ се). Позволява добавяне / вмъкване / премахване / търсене на елементи:

* Add(element) – добавя елемент към List<T>
* Count – връща броя на елементите в List<T>
* Remove(element) – премахва първото срещане на елемент (връща true / false)
* RemoveAt(index) – премахва елемент по неговия индекс
* Insert(index, element) – вмъква елемент на зададената позиция
* Contains(element) – определя дали елемента се съдържа в списъка
* Sort() – сортира във възходящ ред
* Reverse() – обръща списъка наобратно

**Дебъгването** играе важна роля в процеса на създаване на софтуер, която ни позволява постъпково да проследим изпълнението на нашата програма. С помощта на тази техника можем да следим стойностите на локалните променливи, тъй като те се променят по време на изпълнение на програмата, и да отстраним евентуални грешки (бъгове).

**Процесът на дебъгване** на програма включва:

* Откриване на грешка
* Откриване на редовете в кода, които я предизвикват
* Коригиране на грешката в кода
* Проверка дали грешката е отстранена и дали междувременно не са добавени нови грешки
* Това е многократен и продължителен процес

Visual Studio ни предоставя вграден дебъгер (debugger), чрез който можем да поставяме точки на прекъсване (или breakpoints), на избрани от нас места. При среща на стопер (breakpoint), програмата спира изпълнението си и позволява постъпково изпълнение на останалите редове. Дебъгването ни дава възможност да вникнем в детайлите на програмата и да видим къде точно възникват грешките и каква е причината за това.

**Символният низ** е последователност от символи, записана на даден адрес в паметта. Декларират се с ключовата дума string. Символните низове се заграждат в кавички

Класът System.String позволява обработка на символни низове в C#. За декларация на низовете ще продължим да използваме служебната дума string, която е псевдоним (alias) в C# на класа System.String от .NET Framework. Работата със string ни улеснява при манипулацията на текстови данни: построяване на текстове, търсене в текст и много други операции. Пример за декларация на символен низ: string greeting = "Hello, C#";

Класът string има важна особеност – последователностите от символи, записани в променлива от класа, са неизменими (immutable). След като е веднъж зададено, съдържанието на променливата не се променя директно – ако опитаме да променим стойността, тя ще бъде записана на ново място в динамичната памет, а променливата ще започне да сочи към него.

За да може да обработваме декларираната стрингова променлива, трябва да я създадем и инициализираме. Създаването на променлива на клас (познато още като инстанциране) е процес, свързан със заделянето на област от динамичната памет. Преди да зададем конкретна стойност на символния низ, стойността му е null. Това може да бъде объркващо за начинаещия програмист: неинициализираните променливи от типа string не съдържат празни стойности, а специалната стойност null – и опитът за манипу­лация на такъв стринг ще генерира грешка (изключение за достъп до липсваща стойност NullReferenceException)!

Можем да инициализираме променливи по 3 начина:

1. Чрез задаване на низов литерал.

2. Чрез присвояване стойността от друг символен низ.

3. Чрез предаване стойността на операция, връщаща символен низ.

Операции върху символни низове: Сравняване на низове по азбучен ред, Сравнение за еднаквост, Долепване на низове (конкатенация), Търсене на низ в друг низ, Извличане на част от низ.

**Речниците** са известни още като асоциативни масиви (associative arrays) или карти (maps). При речниците заедно с данните, които държим, пазим и ключ, по който ги намираме. Елементите на речниците са двойки (ключ, стойност), като ключът се използва при търсене.

При структурата речник ключът може да не е просто номерче, а всякакъв друг обект. В случая, когато имаме ключ (номер), можем да реализираме такава структура като обикновен масив. Тогава множеството от ключове е предварително ясно – числата от 0 до n, където n е размерът на масива (естествено при разумно ограничение на n). Целта на речниците е да ни освободи, до колкото е възможно, от ограниченията за множеството на ключовете.

При речниците обикновено множеството от ключове е произволно мно­жество от стойности, примерно реални числа или символни низове. Един­ственото задължително изискване е да можем да различим един ключ от друг. След малко ще се спрем по-конкретно на някои допълнителни изисквания към ключовете, необходими за различните реализации.

Речниците съпоставят на даден ключ дадена стойност. На един ключ може да се съпостави точно една стойност. Съвкупността от всички двойки (ключ, стойност) съставя речника.

В програмирането абстрактната структура данни "речник" представлява съвкупност от наредени двойки (ключ, стойност), заедно с дефинирани операции за достъп до стойностите по ключ. Алтернативно тази структура може да бъде наречена още "карта" (map) или "асоциативен масив" (associative array).

Задължителни операции, които тази структура дефинира, са следните:

- void Add(K key, V value) – добавя в речника зададената наредена двойка. При повечето имплементации на класа в .NET, при добавяне на ключ, който вече съществува в речника, се хвърля изключение.

- V Get(K key) – връща стойността по даден ключ. Ако в речника няма двойка с такъв ключ, метода връща null, или хвърля изключение, според конкретната имплементация на речника

- bool Remove(key) – премахва стойността за този ключ от речника. Освен това връща дали е премахнат елемент от речника.

Ето и някои операции, които различните реализации на речници често предлагат:

- bool Contains(key) – връща true, ако в речникът има двойка с дадения ключ.

- int Count – връща броя елементи в речника.

Други операции, които обикновено се предлагат, са извличане на всички ключове, стойности или наредени двойки ключ-стойност, в друга структура (масив, списък). Така те лесно могат да бъдат обходени чрез цикъл.

Функционалност:

* Count – пази броя на двойките от ключ-стойност
* Keys – съдържа уникалните ключове
* Values – съдържа всички стойности
* Основни операции: Add(), [], Remove(), Clear()

Намиране на ключ / стойност:

ContainsKey() – проверяваме дали даден ключ съществува в речника (бърза операция)

ContainsValue() – проверяваме дали дадена стойност съществува в речника (бавна операция)

TryGetValue() – проверяваме дали даден ключ съществува в речника и отпечатва стойността му

Реализацията с хеш-таблица има важното предимство, че времето за достъп до стойност от речника, при правилно използване, теоретично не зависи от броя на елементите в него.

За сравнение да вземем списък с елементи, които са подредени в случаен ред. Искаме да проверим дали даден елемент се намира в него. В най-лошия случай, трябва да проверим всеки един елемент от него, за да дадем категоричен отговор на въпроса "съдържа ли списъкът елемента или не". Очевидно е, че броят на тези сравнения зависи (линейно) от броят на елементите в списъка.

При хеш-таблиците, ако разполагаме с ключ, броят сравнения, които трябва да извършим, за да установим има ли стойност с такъв ключ, е константен и не зависи от броя на елементите в нея. Как точно се постига такава ефективност ще разгледаме в детайли по-долу.

Когато реализациите на някои структури от данни ни дават време за достъп до елементите й, независещ от броя на елементите в нея, се казва, че те притежават свойството random access (свободен достъп). Такова свойство обикновено се наблюдава при реализации на абстрактни струк­тури от данни с хеш-таблици и масиви.

Размерът на таблицата (масива), наричаме капацитет (capacity) на хеш-таблицата. Степен на запълненост (load factor), наричаме реално число между 0 и 1, което съответства на отношението между броя на запълнените елементи и текущия капаците