



# Strukturované programování

**Petr Voborník**

Tento materiál vznikl v rámci realizace projektu  
Strategický rozvoj Univerzity Hradec Králové, reg. č. CZ.02.2.69/0.0/0.0/16\_015/0002427



EVROPSKÁ UNIE  
Evropské strukturální a investiční fondy  
Operační program Výzkum, vývoj a vzdělávání



MINISTERSTVO ŠKOLSTVÍ,  
MLÁDEŽE A TĚLOVÝCHOVY





# Obsah

<b>1</b>	<b>Základní příkazy a algoritmické konstrukce .....</b>	<b>1</b>
1.1	Přehled základních příkazů C# .....	2
1.1.1	Příkazy .....	2
1.1.2	Operátory .....	4
1.1.3	Datové typy .....	6
1.1.4	Garbage collector .....	9
1.1.5	Metody .....	10
1.2	Hello World! .....	13
1.3	Načítání číselných hodnot .....	15
1.4	Vypisování hodnot .....	18
1.4.1	Formátování parametrů v textovém řetězci .....	19
1.5	Operace s celými čísly .....	21
1.6	Hledání extrému .....	22
1.6.1	Větší ze dvou hodnot .....	22
1.6.2	Největší ze tří hodnot .....	22
1.6.3	Největší z více hodnot .....	23
1.7	Prohození dvou proměnných .....	25
1.8	Cyklický posun hodnot .....	27
1.8.1	Cyklický posun vpřed .....	27
1.8.2	Cyklický posun vzad .....	29
1.8.3	Fronta a zásobník .....	31
1.9	Cykly .....	32
1.9.1	Cyklus s podmínkou na začátku .....	32
1.9.2	Cyklus FOR .....	34
1.9.3	Opakované zadávání hesla .....	36
1.9.4	Fibonacciho posloupnost .....	38



<b>2</b>	<b>Vícehodnotové proměnné .....</b>	<b>41</b>
2.1	Posloupnosti (1D pole) .....	41
2.1.1	Základní operace s posloupnostmi .....	41
2.1.2	Výpočty v posloupnosti .....	45
2.1.3	Extrémy .....	46
2.1.4	Změny struktury posloupnosti .....	48
2.2	Matice (2D pole).....	51
2.2.1	Deklarace dvourozměrného pole.....	51
2.2.2	Naplnění matice .....	52
2.2.3	Výpis matice .....	53
2.2.4	Výpočty v maticích .....	54
2.2.5	Čtvercové matice .....	55
2.3	Seznamy .....	56
2.4	Slovníky .....	58
2.5	LINQ.....	62
2.5.1	Agregační funkce.....	63
2.5.2	Pod části seznamu.....	64
2.5.3	Jeden prvek ze seznamu .....	68
2.5.4	Spojování seznamů .....	70
2.5.5	Praktické příklady s LINQ .....	71
<b>3</b>	<b>Rekurzivní zpracování vícerozměrného prostoru .....</b>	<b>74</b>
3.1	Generátor bludiště s „přepážkami“ .....	74
3.2	Vykreslování přepážkové mapy bludiště do konzole .....	80
3.3	Hlavní řídicí program .....	82
3.4	Přidání smyček .....	83
3.5	Nalezení nejvzdálenější buňky .....	92
3.6	Konverze na blokové bludiště .....	98
3.7	Vykreslování blokové mapy do konzole.....	102

# 1 Základní příkazy a algoritmické konstrukce

Čisté strukturované programování, tedy programy psané ve stylu příkaz za příkazem, bez jakéhokoli členění do tříd (v Delphi *unit*) či alespoň metod (jinde zvaných též jako *funkce*, *procedury* či (*pod*)*rutiny*), již s výjimkou skriptovacích jazyků, masivně ustupuje objektově orientovaným programovacím postupům. Avšak právě vnitřky těchto metod tvoří části kódu, jež lze ve spoustě případů ještě stále považovat za kód strukturovaný.

A právě na takovéto krátké příklady, jakožto základní algoritmické konstrukce a jejich správné pochopení, se v této knize zaměříme. Programátoři jsou občas nazýváni zedníky 21. století, a naučit se správně skládat cihlu k cihle, tedy příkaz k příkazu, bude naším cílem. Je to zároveň dobrý základ k tomu, aby bylo možné posléze pokračovat na vyšší stupeň tvorby ucelených programů a aplikací, nebo třeba i architekta rozsáhlých systémů, webových portálů či her.

Právě z důvodů nejširšího možného budoucího rozvoje těchto schopností a dovedností, budou i tyto veskrze primitivní konstrukce prezentovány v objektově orientovaném programovacím jazyce C# z rodiny technologií .NET, který má krom maximálního potenciálu pro pokračování v dalším studiu, zároveň i nejširší spektrum možností využitelnosti<sup>1</sup> a nejefektivnější formu zápisu kód (např. díky LINQ). I proto se zde občas nevyhneme použití nějakého ne zcela strukturálního programátorského prvku, jako jsou např. vlastní metody či statické třídy, ale s ohledem na věci příští, bude takto nabytá znalost pouze ku prospěchu.

---

<sup>1</sup> Konzolové aplikace běžící i na Linuxu (.NET Core), klasické (Windows Forms, WPF) i moderní (UWP) Windows aplikace, mobilní aplikace pro všechny platformy (Xamarin.Forms), mikročipy (.NET Micro Framework) a IoT (Core, UWP, Xamarin), webové aplikace (ASP.NET Core), rozsáhlé modulární systémy, hry (Xamarin, Monogame, Unity)...



## 1.1 Přehled základních příkazů C#

V úvodu začneme přehledem základních příkazů, které se v jazyce C# vyskytují. Ty hlavní z nich v první fázi porovnáme s jejich syntaktickým zápisem v českém pseudokódu, používaném programem Algoritmy<sup>2</sup>, který tomuto kurzu předcházel.

### 1.1.1 Příkazy

Příkaz	Algoritmy	C#	Pozn.
Proměnné	x ...	<code>int i; string s = "text"; var i = 7;</code>	1
Vstup	čti(x);	<code>string x = Console.ReadLine();</code>	2
Výstup	napiš("Hello");	<code>Console.WriteLine("Hello");</code>	3
Přiřazení	x := 5;	<code>int x = 5;</code>	4
Větvení neúplné	jestliže x > 7 pak ...;	<code>if (x &gt; 7) ...;</code>	5, 6
Větvení úplné	jestliže x > 7 pak ... jinak ...;	<code>if (x &gt; 7) ...; else ...;</code>	5, 6
Cyklus s podmínkou na začátku	dokud x < 7 opakuj ...;	<code>while (x &lt; 7) ...;</code>	5, 6
Cyklus zná- mého počtu	pro i od 1 do 7 opakuj ...;	<code>for (int i = 1; i &lt;= 7; i++) ...;</code>	5, 7
Komentář	// Jednořádkový { Víceřádkový }	<code>// Jednořádkový /* Víceřádkový */</code>	8

---

<sup>2</sup> <http://algds.cronos.cz>

## Poznámky

1) Při prvním použití proměnné v kódu je třeba určit, jakého datového typu budou. V rámci této deklarace je také možné jim rovnou nastavit výchozí hodnotu, a jelikož tato hodnota (či proměnná) je sama o sobě nositelem informace, jakého je datového typu, lze místo názvu tohoto typu uvést klíčové slovo **var** (*variable*), a tento typ proměnné kompilátor odvodí z kontextu sám.

2) Metoda pro čtení hodnoty ze vstupního řádku vrací text. Má-li být tento převeden na číselnou hodnotu, je třeba použít konverzi:

```
int cislo = Convert.ToInt32(Console.ReadLine());
```

3) Při vypisování více než jedné hodnoty již nelze použít více parametrů oddělených čárkou (např. `napiš("x = ", x);`), ale buď je do textu zařadit prostřednictvím parametrů (viz kap. 1.4), nebo sloučit tyto hodnoty do jedné, pomocí sčítání textových řetězců:

```
Console.WriteLine("x = " + x);
```

Číselné hodnoty jsou nejprve převedeny na text a sloučeny se zbytkem textu. Mají-li být čísla nejprve sečtena jako čísla a až pak přidána k textu, je třeba je dát do závorky:

```
Console.WriteLine("x + y = " + (x + y));
```

4) Přiřadit hodnotu lze i více proměnným současně jediným příkazem:

```
(a, b, c) = (1, x+3, 55)
```

5) Podmínka musí být vždy ohraničena kulatými závorkami.

6) Pokud má příkaz větvení či cyklu obsahovat více než jeden příkaz, musí být tyto jeho pod-příkazy uvozeny do bloku, jehož začátek a konec je vymezen složenými závorkami:

```
if (x > 7)
{
    ...;
    ...;
}
```



- 7) Příkaz cyklu **for** v kulaté závorce obsahuje tři příkazy oddělené středníkem: deklaraci proměnné počítadla, podmínku rozhodující o tom bude-li cyklus pokračovat nebo skončí a příkaz zvyšující počítadlo v každém kroku cyklu. Každou z těchto tří částí lze v případě potřeby modifikovat (viz kap. 1.9.2).
- 8) Značku dvojitého lomítka lze ve Visual Studiu před každý řádek označené části kódu vložit i zrušit hromadně. Vložení zajišťuje klávesová zkratka **Ctrl+K,C** a zrušení komentáře **Ctrl+K,U**.

### 1.1.2 Operátory

Příkaz	Algoritmy	C#	Pozn.
Celočíselné dělení	<code>a := b div c;</code>	<code>a = b / c;</code>	1
Zbytek po dělení	<code>a := b mod c;</code>	<code>a = b % c;</code>	2
Porovnání	<code>jestliže (a = b and c &lt;&gt; d) ...;</code>	<code>if (a == b &amp;&amp; c != d) ...;</code>	
Logické A	<code>jestliže (a &gt; 10 and a &lt;= 20) pak ...;</code>	<code>if (a &gt; 10 &amp;&amp; a &lt;= 20) ...;</code>	3
Logické NEBO	<code>jestliže (a &lt;= 10 or a &gt; 20) pak ...;</code>	<code>if (a &lt;= 10    a &gt; 20) ...;</code>	4
Inkrementace	<code>i = i + 1;</code>	<code>i++; ++i;</code>	5
Podmíněný operátor	<code>jestliže a &gt; 7 pak   x := 1 jinak   x := 0;</code>	<code>int x = a &gt; 7 ? 1 : 0;</code>	6



## Poznámky

- 1) Aby šlo o celočíselné dělení (**div**) musí být dělitel **c** typu celé číslo (např. **int**), což je i přímo zapsané celé číslo bez desetinné tečky (např. `a = b / 2;`). Pokud je dělitel desetinného typu (např. **double**), je třeba jej přetypovat na celé číslo (např. `a = b / (int)c;`). Bude-li dělitelem desetinné číslo, ať již v proměnné nebo přímo zapsané (např. 2.5 nebo i 2.0), dělení proběhne klasicky matematicky a výsledkem (**a**) tak bude také desetinné číslo, téhož typu jako dělitel.
- 2) Zbytek po celočíselném dělení je např. pro  $7 \% 3 = 1$  nebo  $13 \% 5 = 3$ .
- 3) Znak *ampersand* **&** se na české klávesnici napíše při stisku klávesy pravého **Alt** a písmena **C**.
- 4) Znak svislé čáry **|** se na české klávesnici napíše při stisku klávesy pravého **Alt** a písmena **W**.
- 5) Jako samostatný příkaz je zápis se dvěma plusy před proměnnou nebo za ní ekvivalentní. Rozdíl je, pokud je tento příkaz použit zároveň pro čtení hodnoty proměnné. Pokud je např. `i=5`, pak v obou případech bude po tomto příkazu v **i** hodnota 6. Obalíme-li však obě verze příkazu např. příkazem pro výpis do konzole, tak při `i++` se vypíše 5 (nejdříve se hodnota přečte a až pak zvýší), ale při `++i` se vypíše 6 (nejprve se hodnota zvýší a až pak přečte).
- 6) Výsledkem podmíněného operátoru **?:**, je jedna z hodnot před či za dvojtečkou. Která z nich to bude určí platnost podmínky před otazníkem *podmínka ? hodnota\_když\_platí : hodnota\_když\_neplatí*. Takovýto výraz lze přímo použít i jako součást složitějšího výpočtu (např. `prumer = soucet / (pocet != 0 ? pocet : 1)`) a lze jej také použít opakovaně (např. `max = a > b && a > c ? a : b > c ? b : c`). Ternární podmíněný operátor není výsadou C#, ale používá se i v mnoha dalších programovacích jazycích (např. C, C++, Java, PHP, JavaScript apod.).

### 1.1.3 Datové typy

Jak již bylo uvedeno dříve, každá proměnná v C# musí mít jasně definovaný svůj datový typ hned při svém prvním použití v kódu. Jaké jsou dostupné standardní hodnotové datové typy ukazují následující tabulky. Nejprve tedy datové typy pro **celá čísla**.

Název	Typ	Rozsah	Rozsah (přesný)	Velikost
sbyte	SByte	$-2^7$ až $2^7-1$	-128 až 127	1 B
short	Int16	$-2^{15}$ až $2^{15}-1$	-32 768 až 32 767	2 B
int	Int32	$-2^{31}$ až $2^{31}-1$	-2 147 483 648 až 2 147 483 647	4 B
long	Int64	$-2^{63}$ až $2^{63}-1$	-9 223 372 036 854 775 808 až 9 223 372 036 854 775 807	8 B
byte	Byte	0 až $2^8-1$	0 až 255	1 B
ushort	UInt16	0 až $2^{16}-1$	0 až 65 535	2 B
uint	UInt32	0 až $2^{32}-1$	0 až 4 294 967 295 (cca 4,3 mld.)	4 B
ulong	UInt64	0 až $2^{64}-1$	0 až 18 446 744 073 709 551 615	8 B
-	BigInteger	$-\infty$ až $\infty$	není klasický typ, max. rozsah dle volné paměti	

Jak je v tabulce vidět, jednotlivé datové typy se liší jednak rozsahem hodnot, kterých mohou nabývat, ale také bytovou velikostí, kterou zabírají v operační paměti během svého používání. Proto není příliš vhodné vždy používat pouze ten největší, ale volit je s rozmyslem tak, aby jejich rozsah stačil na účel, který mají plnit a zároveň zabíral v paměti co nejmenší prostor. Rozumným kompromisem a zároveň výchozím datovým typem pro celá čísla zapsaná v kódu je typ **int** (*integer* – celé číslo), pravým názvem struktury **Int32**, tzn. s celkovým počtem možných hodnot  $2^{32}$ .

Pokud je třeba použít čísel **desetinných**, pak jejich základní typy ukazuje následující tabulka.

Název	Typ	Rozsah	Číslic	Velikost
float	Single	$+(-) 1.5 \cdot 10^{-45}$ až $+(-) 3.4 \cdot 10^{38}$	7	4 B
<b>double</b>	Double	$+(-) 5.0 \cdot 10^{-324}$ až $+(-) 1.7 \cdot 10^{308}$	15-16	8 B
decimal	Decimal	$+(-) 1.0 \cdot 10^{-28}$ až $+(-) 7.9 \cdot 10^{28}$	28-29	16 B

U desetinných čísel není až tak podstatný rozsah, ale spíše počet číslic, se kterými dokážou pracovat. Mezi nimi je pak na dané pozici umístěna desetinná čárka. Čím více cifer má celá část čísla, tím méně jich zbude na ty za čárkou a tím bude tedy i nižší přesnost celého čísla. Výchozím typem je v tomto případě typ **double** (tzv. *dvojitá přesnost* oproti té nejnižší *float*).

U desetinných čísel je nezbytné mít na paměti fakt, že pro jejich uchování v paměti je použita binární hodnota (např. pro double 8 bajtů po 8 bitech čili 64 bitů, tj. 64 nul či jedniček), nikoli textová podoba čísla. Hodnoty jsou tak sice uchovávány dosti přesně, avšak ne pro každou z nich je možné mít přesnou binární obdobu, a proto na posledních editovatelných místech za desetinnou čárkou dochází k drobným nepřesnostem.

Pokud jsou pak čísla vypisována pouze s určitým počtem desetinných míst a při ostrém porovnávání zaokrouhlována, je vše v pořádku. Porovnáme-li ale proměnnou desetinného typu s nějakou konkrétní hodnotou, až na pár výjimek (např. 0) se hodnoty rovnat nebudou. Výstupem následujícího příkladu tak bude „ne“.

```
1: double x = 0.1 + 0.2;
2: if (x == 0.3)           // false
3:     Console.WriteLine("ano");
4: else
5:     Console.WriteLine("ne");
```

Pokud bychom ale při porovnání (ř. 2) hodnotu **x** zaokrouhlili třeba i na 15 desetinných míst (`Math.Round(x, 15) == 0.3`), byl by výsledek „**ano**“.

S jakými přesně hodnotami je ve skutečnosti v paměti počítáno, ukazuje následující příklad. V něm je použit vědecký (přesný) formát výpisu čísla (**G**) s daným počtem desetinných míst (viz kap. 1.4.1). Výsledek výpisu je pak u každého příkazu uveden v komentáři za ním.

```
1: Console.WriteLine("{0:G17}", 0.1 + 0.2); // 0,300000000000000004
2: Console.WriteLine("{0:G17}", 0.3);       // 0,29999999999999999
3: Console.WriteLine("{0:G16}", 0.3);       // 0,3
```

Méně desetin tedy v určitých případech může znamenat více přesnosti. Toto nicméně není specifikum jazyka C#, ale téměř všech programovacích jazyků<sup>3</sup>.

Další často používané datové typy jsou uvedeny v poslední tabulce.

Název	Typ	Popis	Velikost
<b>bool</b>	Boolean	Logická hodnota <b>ano</b> (true) nebo <b>ne</b> (false)	teoreticky 1 b reálně 1 B
char	Char	1 znak v kódování Unicode (např. 'c')	16 b = 2 B
<b>string</b>	String	Textový řetězec (např. "text")	podle délky

Textový řetězec (**string**) je zároveň jednorozměrným polem (posloupností) znaků (**char**). Zápis konstantních hodnot obou těchto typů v kódu je rozlišen tak, že textový řetězec je zapsán mezi uvozovky, zatímco znak mezi apostrofy.

---

<sup>3</sup> viz <https://0.300000000000000004.com>

Veškeré typy uvedené v těchto tabulkách (kromě **string**), jsou tzv. hodnotové datové typy (*struktury*), což jednak znamená, že se jejich hodnota při přiřazení do další proměnné v paměti kopíruje (toto platí i pro **string**), a také musí mít vždy přiřazenu nějakou hodnotu (nemohou být **null**). Pokud ale potřebujeme nějaké proměnné takového typu umožnit být zároveň **null** (např. ve smyslu, že tato informace není známa, podobně jako u databázi), pak stačí při deklaraci takové proměnné za datový typ připsat otazník (např. **int?**), což je zkratka pro zápis **Nullable<int>** (obě varianty jsou ekvivalentní). Díky tomu tedy např. proměnná typu **bool?** může nabývat tří hodnot (**true**, **false** a **null**).

#### 1.1.4 Garbage collector

Jakmile proměnnou deklarujeme, vyhradí se pro ni místo v paměti odpovídající jejímu datovému typu. Do tohoto místa adresovaného přes název proměnné můžeme hodnoty daného datového typu ukládat a číst je. Co se ale s touto částí paměti stane, když už proměnnou nepotřebujeme?

C# je, mimo jiné, vybaven jednou velmi užitečnou funkcí, která se jmenuje „Garbage collector“, česky „sběratel odpadků“ nebo taky jednoduše „popelář“. Tato funkce, zjednodušeně řečeno, neustále hlídá používané proměnné a pokud jsou tam nějaké, které v dalším kódu již nebudou použity (ať již pro zápis nebo pro jen čtení), uvolní jejich místo v paměti, aby jej mohly používat jiné proměnné, které budou v kódu teprve deklarovány, nebo třeba i jiné programy.



Toto uvolňování paměti již dále nepoužívaných proměnných s přímo nevymezenou platností (např. lokální proměnné metody) přitom neprobíhá neustále, ale pouze ve chvílích, kdy je to buď z důvodů nedostatku paměti potřeba, anebo když má aplikace pro takovéto úkony volný výpočetní čas.

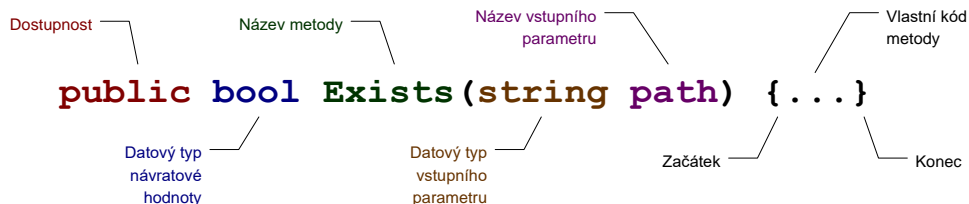
Hodnoty uložené v paměti proměnných jsou kvůli bezpečnosti samozřejmě před jejich předáním k volnému používání vynulovány.

Díky tomu lze v kódu jazyka C# používat a deklarovat libovolné množství proměnných a není nezbytné (až na extrémní případy) řešit, od kdy do kdy je vlastně budeme používat a dalším „nadbytečným“ kódem je uvolňovat z paměti. To za vás mnohem lépe a vyřeší právě Garbage collector.

V jiných, spíše starších, programovacích technologiích, se třeba proměnné uvolňovaly až při vypnutí programu, takže paměť, kterou déle spuštěný program zabíral neustále narůstala až mnohdy do neúnosných rozměrů. V jiných jazycích (např. C++) je zase v kódu nezbytné pro každou proměnnou volat i speciální příkaz pro její uvolnění, což sice může být pro práci s pamětí efektivnější, ale velmi to prodlužuje a nezpřehledňuje zdrojový kód a zapomenuté či naopak předčasné uvolnění nějaké proměnné je velmi častou příčinou pádu těchto aplikací. Oproti tomu „Garbage collector“ má například také starší Java.

### 1.1.5 Metody

Metody, dříve též zvané *procedury* (nic nevrací) nebo *funkce* (vracejí nějaký výsledek), jsou samostatné části kódu, které lze pod jejich názvem volat (spouštět) opakovaně. Struktura jejich deklarace je následující.



Metody jsou vždy umístěny v rámci nějaké třídy (v našem případě **Program**). Dostupnost definuje možnost použití metody ostatními třídami.

Není-li žádná z možností (ty hlavní jsou **public**, **private** a **protected**) uvedena, znamená to, variantu **private** (soukromá), tedy že metodu bude možné spouštět pouze z kódu třídy v jejímž rámci se metoda nachází.

Datový typ návratové hodnoty určuje, jakého typu bude hodnota, jež metoda vrací jako výsledek své činnosti (klíčovým slovem **return**). Pokud se místo konkrétního datového typu uvede klíčové slovo **void**, znamená to, že metoda žádnou hodnotu nevrací.

```
1: /// <summary>Vrátí aritmetický průměr ze dvou hodnot</summary>
2: public static double Prumer(int a, int b)
3: {
4:     return (a + b) / 2.0;
5: }
```

Mezi dostupností a typem návratové hodnoty ještě může být uvedeno klíčové slovo **static**. Jeho přítomnost určuje, že metodu lze spouštět bez nutnosti vytvoření instance třídy, čemuž se v této knize zatím budeme vyhýbat.

Vstupních parametrů metody může být libovolný počet. Jejich seznam se zapisuje do kulatých závorek za název metody (pokud metoda žádné parametry nemá, pak jsou za názvem metody pouze prázdné kulaté závorky) a jejich jednotlivé definice se oddělují čárkami. U každého parametru zvlášť musí být vždy uveden nejprve jeho datový typ a poté název, pod kterým bude v kódu metody dostupný podobně jako klasické proměnné.

Parametry metody nemusí být jen vstupní, ale i výstupní (popř. referenční). Před těmi výstupními se ještě před datový typ uvádí klíčové slovo **out**.



```
1: /// <summary>Vydělí dvě hodnoty a informuje o výsledku</summary>
2: /// <param name="podil">Výsledek dělení, povedlo-li se</param>
3: /// <returns>Povedlo se hodnoty vydělit? </returns>
4: static bool Deleni(int delenec, int delitel, out double podil)
5: {
6:     if (delitel == 0)
7:     {
8:         podil = double.NaN; // Musí být nastaveno před return
9:         return false; // Po return už kód metody nepokračuje
10:    }
11:    podil = delenec / (double)delitel;
12:    return true;
13: }
```

Všem výstupním parametrům musí být vždy nastaveny hodnoty ještě před koncem metody (tj. koncem jejího kódu nebo jakýmkoli příkazem **return**, který případný další kód přeskakuje).

Zvláštní formát komentáře nad záhlavím metody (tři lomítka, element **<summary>** apod.) umožňují to, že při zapisování názvu metody pro její volání z jiné části kódu, se text tohoto komentáře zobrazí jako nápovědná bublina pro tuto metodu (popř. pro její jednotlivé parametry a výstup).



## 1.2 Hello World!

S ucelenými příklady začneme klasickým nejjednodušším „programem“, kterým se obvykle nové prostředí testuje – programem „Hello World“. Ten má za úkol vypsat pouze tento text na obrazovku tak, abychom si jej mohli přečíst, načež může skončit.

U konzolových aplikací spouštěných ve Windows v samostatném okně, pokud si chceme vypsaný text prohlédnout, je třeba přidat ještě příkaz, který program těsně před jeho koncem pozdrží, aby se konzolové okno hned nevypnulo (ř. 6). Kód programu pak může vypadat následovně.

```
1: class Program
2: {
3:     static void Main(string[] args)
4:     {
5:         Console.WriteLine("Hello World!");
6:         Console.ReadLine(); // Počká s koncem na stisk Enteru
7:     }
8: }
```

Tento příklad ukazuje vlastní kód programu (ř. 5–6) zapsaný v kontextu celé třídy **Program** a její statické metody **Main** (jejich hlavička je vygenerována automaticky při založení nového konzolového projektu). Ta je totiž tou výchozí metodou každého konzolového programu, jejíž kód je automaticky spouštěn. V následujících příkladech již toto „okolí“ znovu zmiňováno nebude a nebude-li uvedeno jinak, umístění kódu každého příkladu bude myšleno právě do této metody **Main**.

Pokud tento klasický příklad rozšíříme tak, aby místo „světa“ zdravil osobu jména, které před tím zadá uživatel, může kód vypadat třeba následovně.



```
1: // Vypíše řádek (bez jeho zalomení na konci)
2: Console.Write("Zadej své jméno: ");
3: // Načte text (jméno) od uživatele (zadávání je zakončeno
   klávesou Enter)
4: string jmeno = Console.ReadLine();
5: // Vypíše řádek s pozdravem zadaného jména (a zalomí řádek)
6: Console.WriteLine("Ahoj " + jmeno + "!");
7: // Složení textu pomocí indexovaných parametrů
8: Console.WriteLine("Ahoj {0}!", jmeno);
9: // Složení textu pomocí přímo vložených parametrů ($ před ")
10: Console.WriteLine($"Ahoj {jmeno}!");
```

V tomto příkladu je tedy uživatel vyzván k zadání jména, které se načte do proměnné **jmeno** a to je posléze vypsáno v kombinaci se slovem "Ahoj " (s mezerou na konci, aby jméno nebylo k tomuto slovu „nalepeno“) zleva a vykřičníkem zprava.

Toto vypsání je v příkladu provedeno hned 3x, přičemž výstup je ve všech případech naprosto stejný. Rozdíl je pouze v zápisu tohoto výstupu v kódu, a to pomocí skládání textových řetězců za sebe (ř. 6), pomocí indexovaných parametrů (ř. 8) a pomocí parametrů vložených přímo do textového řetězce (ř. 10), kde musí být před uvozovkami znak dolaru \$.

Nejuniverzálnější z těchto tří možností je ta druhá (pomocí indexovaných parametrů, ř. 8), jelikož jako jediná v případě celých vět umožňuje jejich lokalizaci, tzn. překlad do cizích jazyků, kde může být pořadí slov různé. Zároveň stejně funguje i metoda **Format** třídy **String**, takže stejným způsobem lze textové řetězce kombinovat nejen při výpisu do konzole, ale v podstatě při jakékoli příležitosti. Tuto variantu tedy budeme v následujícím kódu používat nejčastěji.



### 1.3 Načítání číselných hodnot

Načtení hodnoty typu **string** (textový řetězec) ukazoval již předchozí příklad (ř. 4). Co když ale potřebujeme načíst jiný typ hodnoty do proměnné, např. celé číslo (**int**)? Několik možností, jak na to, ukazují následující příklady.

```
1: // Načtení číselné hodnoty bez kontroly validity vstupu
2: string sa = Console.ReadLine(); // Načte textový řetězec
3: int a = Convert.ToInt32(sa);    // Převede text na číslo
4:
5: // Načtení číselné hodnoty "najednou"
6: int b = Convert.ToInt32(Console.ReadLine());
```

První dva způsoby jsou totožné, přičemž první je jen rozepsán na dva samostatné příkazy (ř. 2–3), a ve druhém je obojí zapsáno najednou (ř. 6). Oba tyto způsoby ale napevno počítají s tím, že uživatel zadá platné celé číslo (např. 7 nebo -145), ovšem pokud bude zadán nečíselný text (např. „jedna“ nebo „7x“), konverze textu na číslo selže a vyvolá tzv. výjimku, která pokud není ošetřena způsobí zaseknutí programu.

Kód, který vyjimku očekává a tzv. ji „ošetří“ ukazuje další příklad.

```
1: // Načtení číselné hodnoty se zachycením případné výjimky
2: try
3: {
4:     int d = Convert.ToInt32(Console.ReadLine());
5:     // d je nastaveno - převod na číslo byl úspěšný
6: } catch {
7:     // d není nastaveno - převod na číslo se nezdařil
8: }
```

Pokud tedy bude zadáno platné číslo (ř. 4), vše by mělo v pořádku pokračovat v bloku kódu **try** (ř. 5). Jestliže zde však dojde k chybě, vykonávání kódu

přeskočí do části **catch** (ř. 7). Ať už se ale stane cokoli kód bude následně pokračovat za touto konstrukcí dál (za ř. 8), aniž by se uživateli zobrazilo jakékoli chybové hlášení, jako by se nic nestalo.

S proměnnou **d** však bude možné pracovat pouze uvnitř bloku **try**, protože právě zde byla deklarována. Pokud bychom s ní chtěli pracovat i za tímto blokem, musela by se deklarovat a nějak nastavit ještě před tímto blokem (před ř. 2), např. `int d = 0;`. V případě chyby při konverzi by v proměnné **d** pak tato přednastavená hodnota zůstala.

Třetí varianta je šetrnější a stejného výsledku dosahuje i bez vyvolávání výjimky, která i když je ošetřena, je na celý proces zpracování mnohem náročnější.

```
1: // Načtení číselné hodnoty s kontrolou validity
2: string sc = Console.ReadLine();
3: if (int.TryParse(sc, out int c))
4: {
5:     // c je nastaveno - převod na číslo byl úspěšný
6: } else {
7:     // c není nastaveno - převod na číslo se nezdařil
8: }
```

Metoda **TryParse** (ř. 3) struktury **int** (popř. **Int32**), má jako první vstupní parametr textový řetězec, který se pokusí převést na číslo. Druhým, tentokrát výstupním parametrem (uvozeno modifikátorem parametru **out** a s možností deklarace přímo zde, viz kap. 1.1.5), bude převedené číslo. To, jestli se převod na číslo zdařil nebo ne, vrací tato metoda jako svou výstupní hodnotu typu **bool**, proto ji lze takto rovnou zařadit do podmínky (**if**).

Proměnná **c** je v tomto případě deklarována a nastavena v každém případě, avšak pouze v kladné větvi této podmínky (ř. 5) byl převod úspěšný a lze tak

pracovat se správnou hodnotou. V **else** větvi (ř. 7) pak v proměnné **c** bude defaultní hodnota pro tento typ, nemající nic společného s tím, co zadával uživatel. Za touto konstrukcí (za ř. 8) pak může platit obojí.

Jak zajistit, aby uživatel zadal platné číslo před dalším pokračováním programu, i kdyby se mu to povedlo až na několikátý pokus, ukazuje následující příklad.

```
1: // Zadávání se opakuje, dokud není zadáno platné celé číslo
2: string se = Console.ReadLine();
3: int e;
4: while (!int.TryParse(se, out e))
5:     se = Console.ReadLine();
```

Tentokrát bude v proměnné **e** platné celé číslo zadané uživatelem určitě, protože jinak jej program nepustí dál (za ř. 5). Bylo by však vhodné, přidat do tohoto kódu i vypisování instrukcí pro uživatele, co po něm program vlastně v každém kroku požaduje.



## 1.4 Vypisování hodnot

Základní principy pro výstup neboli vypisování textu do konzole již byl představen v kap. 1.2. U číselných hodnot, popř. výsledku nějakého výpočtu, je postup obdobný, jak ostatně ukazuje následující příklad.

```
1: // Načtení dvounou celých čísel
2: int a = Convert.ToInt32(Console.ReadLine());
3: int b = Convert.ToInt32(Console.ReadLine());
4: // Výpočet výsledku (součet)
5: int c = a + b;
6: // Vypsání výsledné hodnoty součtu
7: Console.WriteLine(c);
8: // Vypsání celého příkladu různými způsoby se stejným výsledkem
9: // (např. pro a=3, b=2 bude výstup: "3 + 2 = 5")
10: Console.WriteLine("{0} + {1} = {2}", a, b, c);
11: Console.WriteLine("{0} + {1} = {2}", a, b, a + b);
12: Console.WriteLine($"{a} + {b} = {c}");
13: Console.WriteLine($"{a} + {b} = {a + b}");
```

Kód opět uvádí vícero způsobů, jak vypsát výsledek (ř. 7), resp. celý příklad (ř. 10–13). První dva způsoby (ř. 10–11) používají parametrický způsob, druhé dva (ř. 12–13) pak parametrů vložených přímo do textového řetězce. U obou variant lze přitom použít nejen způsob vypsání výsledku předem vypočítaného do proměnné *c* (ř. 10 a 12), ale také je možné tento výsledek vypočítat až přímo v rámci příkazu pro výpis (ř. 11 a 13).

Parametrů je zde zároveň více než jeden, takže se mění i jejich indexy. Hodnoty jsou pak dosazovány podle těchto indexů a pořadí, v jakém jsou zadány, jako druhý a další parametry metody **Write** či **WriteLine**.

Takže například příkazy

```
Console.WriteLine("{0} + {1}", 5, 7);
```

```
Console.WriteLine("{1} + {0}", 7, 5);
```

budou mít oba totožný výstup „5 + 7“.

### 1.4.1 Formátování parametrů v textovém řetězci

Parametrický zápis výstupu má ještě další výhody. Indexované parametry v textovém řetězci, za které jsou pak hodnoty dosazované zvlášť, mohou být samy opatřeny dalšími parametry, určujícími formát, v jakém se má hodnota příslušného parametru dosadit. Přehled těchto možností ukazuje následující tabulka.

Parametr	Popis	Výstup
{0}	Klasický parametr bez dodatečného nastavení	„12345“
{0:N0}	N jako <i>number</i> , tj. formát čísla, bez desetin	„12 345“
{0:N2}	Formát čísla se dvěma desetinnými	„12 345,00“
{0:dd.MM.yyyy}	Formát pro datový typ datum a čas <i>DateTime</i>	„21.03.2019“
{0,2}	Hodnotu doplní zleva mezerami na celkovou šířku 2 znaků	„ 7“
{0,5}	Hodnotu doplní zleva mezerami na celkovou šířku 5 znaků	„ 7“
{0,-5}	Hodnotu doplní zprava mezerami na celkovou šířku 5 znaků	„7 “
{0,5:N1}	Formát čísla, 1 desetina, celková šířka 5 znaků, v případě potřeby doplněno mezerami zleva	„ 7,0“

Za dvojtečkou se tedy dá určit formát čísla, popř. jiné proměnné, kterou lze formátovat (např. **DateTime**). Formát **N** je pouze jednou z možností, jak formátovat číslo a odkazuje se na nastavení formátu čísla v operačním systému. Díky tomu se tedy v české verzi číslo ve formátu **N2** vypíše jako

„12 345,00“ a ve verzi americké „12,345.00“. Alternativami pro **N** jsou např. **C** (currency – měna), **E** (exponencial – matematický zápis s  $10^x$ ), **P** (percent – procenta, kdy 0,75 je 75%), **G** (general - kompaktnější fixní či vědecký zápis) atd.<sup>4</sup>

Pokud za idnex parametru přidáme čárku (což se v případě použití obého musí stát ještě před dvojtečkou), tak za ní uvedená hodnota definuje, kolik znaků má být řetězec dosazovaný za tento parametr dlouhý. Pokud počet znaků s touto hodnotou souhlasí, nebo je delší, pak je hodnota vložena bezeměny. Je-li však počet znaků menší než absolutní verze této hodnoty, tak je řetězec doplněn mezerami na příslušnou délku. V případě kladné hodnoty se mezery doplňují zleva, v případě záporné zprava. Altrnativou jsou metody třídy **string PadLeft** (doplň zleva) či **PadRight** (doplň zprava), kde lze určit nejen minimální délku řetězce, ale i znak pro doplňování (např. místo mezery používat nulu nebo pomlčku).

---

<sup>4</sup> Další standardní formáty čísel viz <https://docs.microsoft.com/en-us/dotnet/standard/base-types/standard-numeric-format-strings>





## 1.5 Operace s celými čísly

Následující příklad ukazuje, základní matematické operace s celými čísly.

```
1: // Načtení dvou celých čísel
2: Console.Write ("a = ");
3: int a = Convert.ToInt32(Console.ReadLine());
4: Console.Write ("b = ");
5: int b = Convert.ToInt32(Console.ReadLine());
6: // Základní výpočty se dvěma čísly
7: Console.WriteLine("Součet = {0}", a + b);
8: Console.WriteLine("Součin = {0}", a * b);
9: Console.WriteLine("Rozdíl = {0}", a - b);
10: // Dělení: přesné, celočíselné a zbytek po celočíselném dělení
11: Console.WriteLine("Podíl = {0:N3}", a / (double)b);
12: Console.WriteLine("Celočíselné dělení = {0}", a / b); // div
13: Console.WriteLine("Zbytek po dělení = {0}", a % b); // mod
14: // Matematické funkce
15: Console.WriteLine("Sin(pi/2) = {0}", Math.Sin(Math.PI / 2.0));
```

Ukázka výstupu v konzolovém okně pak může vypadat následovně.

```
a = 7
b = 4
Součet = 11
Součin = 28
Rozdíl = 3
Podíl = 1,750
Celočíselné dělení = 1
Zbytek po dělení = 3
Sin(pi/2) = 1
```



## 1.6 Hledání extrému

Extrémem je myšlena hodnota, která je v řadě čísel, ať již předem daných, nebo průběžně zadávaných, nejvyšší či naopak nejnižší. Postup hledání je v obou případech (pro maximum i minimum) stejný, pouze stačí otočit znaménko porovnání ( $>$  na  $<$  a naopak).

### 1.6.1 Větší ze dvou hodnot

Pokud máme pouze dvě čísla, stačí je porovnat a větší z nich vypsát jako výsledek. Nazýváme-li však výsledek „větším číslem“, je třeba otestovat také variantu, že jsou obě čísla stejná, protože, pak nelze o žádném z nich tvrdit, že je „větší“.

```
1: // Načtení dvou celých čísel
2: int a = Convert.ToInt32(Console.ReadLine());
3: int b = Convert.ToInt32(Console.ReadLine());
4: // Určení a vypsání větší hodnoty
5: if (a == b)
6:     Console.WriteLine("Čísla jsou stejná {0}", a);
7: else
8:     if (a > b)
9:         Console.WriteLine("Větší je {0}", a);
10:    else
11:        Console.WriteLine("Větší je {0}", b);
```

Pořadí, ve kterém jsou čísla porovnávána lze samozřejmě libovolně změnit (např.  $a > b$ ,  $a < b$ ,  $a == b$ ) a dosáhnout tak stejného výsledku.

### 1.6.2 Největší ze tří hodnot

Pokud porovnáváme alespoň tři hodnoty, nelze již jejich maximum označovat za *větší* číslo, ale je třeba používat výraz *největší*. Toto označení zároveň nenutí rozlišovat případy, kdy je těchto maxim mezi čísly více, takže není-li z podstaty zadání tato informace přímo vyžadována, nemusí být nezbytně programem řešena.

Budeme-li pokračovat v předchozím stylu porovnávání hodnot, pak se můžeme dopracovat k následujícímu kódu.

```
1: // Načtení tří celých čísel
2: int a = Convert.ToInt32(Console.ReadLine());
3: int b = Convert.ToInt32(Console.ReadLine());
4: int c = Convert.ToInt32(Console.ReadLine());
5: // Určení a vypsání největší hodnoty
6: if (a > b)
7:     if (a > c)
8:         Console.WriteLine("Největší je {0}", a);
9:     else
10:        Console.WriteLine("Největší je {0}", c);
11: else
12:     if (b > c)
13:         Console.WriteLine("Největší je {0}", b);
14:     else
15:         Console.WriteLine("Největší je {0}", c);
```

Toto řešení však není již zrovna optimální a s každou další proměnnou by kód mnohonásobně narůstal do neúnosného rozsahu.

### 1.6.3 Největší z více hodnot

Jak tedy určit maximum z většího počtu hodnot? Není třeba porovnávat každé číslo s každým z ostatních, ale stačí nám jedno porovnání na každou z hodnot v řadě. Následující obrázek ukazuje, jak by se dalo posupovat při hledání nejvyššího sněhuláka.



Postup je tedy takový, že si vezmeme prvního sněhuláka v řadě a s jeho výškou porovnáváme ty následující. Pokud narazíme na nějakého většího, tak budeme pokračovat s ním a zbytek řady porovnáváme již s jeho výškou. Se sněhuláky se samozřejmě moc hýbat nedá, abychom je mohli vždy postavit vedle sebe a vidět, který je větší. Naštěstí nám stačí pracovat pouze s jejich výškami, tj. každého musíme změřit a pak stačí porovnávat již pouze tato čísla.

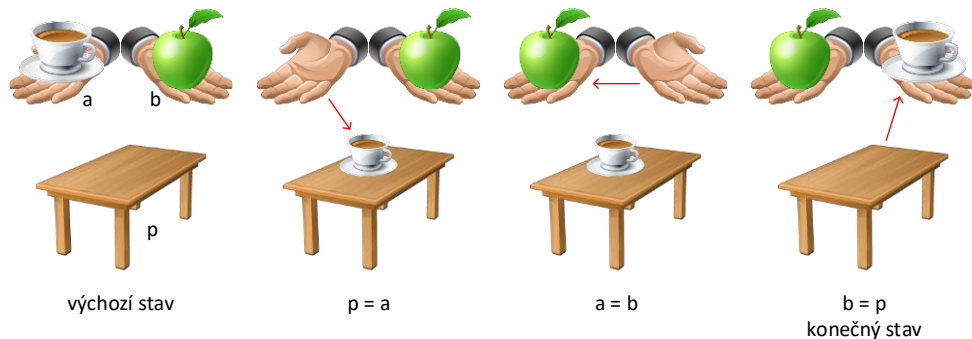
Následující kód ukazuje, jak tímto způsobem určit maximum ze čtyř číselných hodnot.

```
1: // Načtení čtyř celých čísel
2: int a = Convert.ToInt32(Console.ReadLine());
3: int b = Convert.ToInt32(Console.ReadLine());
4: int c = Convert.ToInt32(Console.ReadLine());
5: int d = Convert.ToInt32(Console.ReadLine()); // atd.
6: // Určení a vypsání největší hodnoty
7: int max = a;
8: if (b > max) max = b;
9: if (c > max) max = c;
10: if (d > max) max = d; // stejně lze pokračovat i na dále
11: Console.WriteLine("Největší je {0}", max);
```

Pro hledání zde použijeme pomocnou proměnnou **max**, do které si uložíme hodnotu z první porovnávané proměnné **a** a ty další porovnáváme již pouze s ní. Pokud narazíme na nějakou větší hodnotu, nahradíme dosavadní maximum touto hodnotou a další porovnávání již probíhá s tímto novým maximumem. Na závěr nám v proměnné **max** zůstane největší hodnota ze všech a tu také vypíšeme jakožto výsledek hledání.

## 1.7 Prohození dvou proměnných

Představte si, že máte v jedné ruce šálek kávy (**a**) a ve druhé jablko (**b**). Jak jejich obsah zaměníte? Potřebujete místo, kam si jednu z těchto věcí dočasně odložíte, například stůl (**p**).



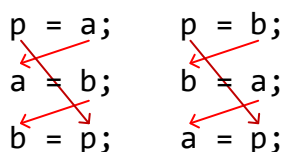
Šálek odložíte z první ruky na stůl ( $p = a$ ), tím si tuto ruku uvolníte a můžete do ní ze druhé ruky předat jablko ( $a = b$ ). Nyní máte volnou druhou ruku a můžete jí tak zvednout jablko ze stolu ( $b = p$ ). Stůl byl před touto operací prázdný a bude i po ní, takže jej v předchozích ani následujících krocích nemusíte nijak dále uvažovat.

Pomocí kódu se pak celá operace prohození hodnot dvou proměnných zapíše následovně.

```
1: // Načtení dvou hodnot do proměnných a a b
2: int a = Convert.ToInt32(Console.ReadLine()); // např. 1
3: int b = Convert.ToInt32(Console.ReadLine()); // např. 2
4: // Prohození dvou proměnných s výpisem jejich hodnot před a po
5: Console.WriteLine("{0}, {1}", a, b);           // např. 1, 2
6: int p = a;
7: a = b;
8: b = p;
9: Console.WriteLine("{0}, {1}", a, b);           // např. 2, 1
```

Jestli na odložení do pomocné proměnné (**p** – na stůl) zvolíme hodnotu první proměnné (**a** – obsah první ruky), nebo druhé proměnné (**b** – obsah druhé ruky) není podstatné. Důležité však je, že v následujícím kroku můžeme přepsat pouze hodnotu té proměnné, kterou jsme si odložili, nikoli obráceně. Při zápisu v kódu pak musí platit pořadí proměnných, tak, jak jej naznačují šipky.

$p = a;$	$p = b;$
$a = b;$	$b = a;$
$b = p;$	$a = p;$



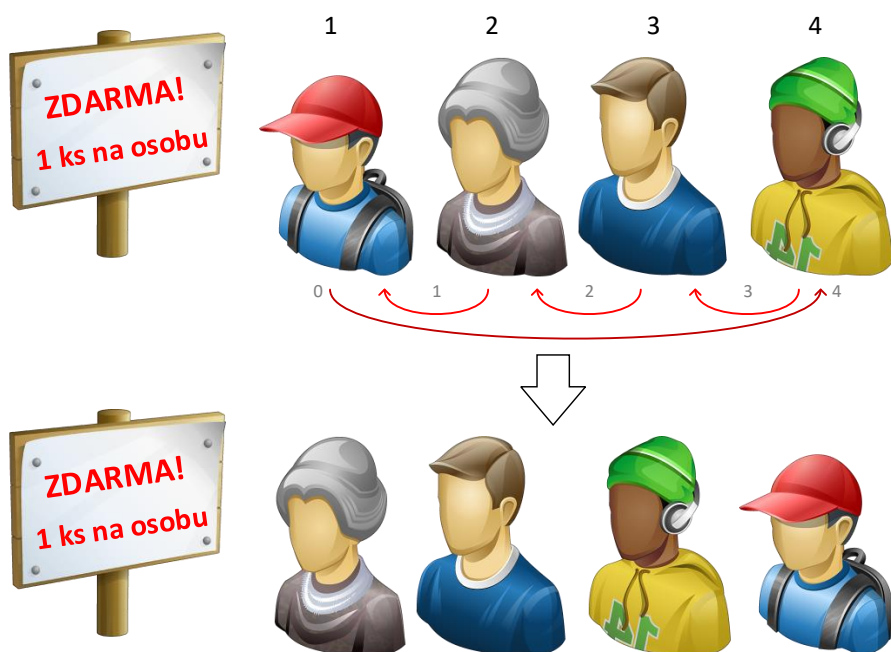
Hodnoty se, na rozdíl od předmětů v reálném světě, mezi proměnnými ve skutečnosti nepřesouvají, ale kopírují. Takže pokud je `a = 3`, tak po příkazu `p = a` bude hodnota 3 zkopírovaná do proměnné **p**, ale zároveň zůstane i v proměnné **a**. Jedná se ale o kopii, takže následná změna hodnoty v kterékoli z těchto dvou proměnných již neovlivní hodnotu v té druhé.

## 1.8 Cyklický posun hodnot

Cyklický posun je obdobou prohození dvou proměnných, pouze proměnné jsou minimálně tři, a to uspořádané v řadě (v tomto případě abecedně dle názvů proměnných). Tentokrát již záleží na pořadí, ve kterém posun provedeme, v základu rozlišujeme dva, a to vpřed a vzad.

### 1.8.1 Cyklický posun vpřed

Cyklický posun vpřed by se při výpisu řady hodnot do jednoho řádku za sebe (např. „1,2,3,4“) také dal označit jako posun *doleva*. Představit si to můžeme třeba jako frontu, kde každý čekající má lístek s číslem, určujícím jeho pořadí. Nejprve odbaví toho s číslem 1, pak s číslem 2 atd. Jelikož je to ale posun *cyklický*, tak se odbavený čekající do fronty vždy hned zase vrátí, a to na její konec (asi dávají něco zdarma). Zároveň do této fronty nepřibývají žádní další čekající. Celou situaci ilustruje následující obrázek.



V obrázku je naznačeno jak pořadí čekajících, tak i pořadí kroků, které je pro jeden takovýto cyklický posun vpřed třeba vykonat. Následující kód ukazuje, jak celý posun zapsat programově se třemi proměnnými.

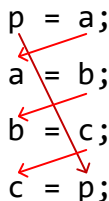
```
1: // Nastavení výchozích hodnot (lze načíst i klasickým způsobem)
2: int a = 1, b = 2, c = 3;
3: // Cyklický posun vpřed/doleva (1, 2, 3 => 2, 3, 1)
4: Console.WriteLine("{0}, {1}, {2}", a, b, c); // 1, 2, 3
5: int p = a; // První (putovní) proměnná se odloží
6: a = b;
7: b = c;
8: c = p;      // Odložená (putovní) proměnná se vrátí do té poslední
9: Console.WriteLine("{0}, {1}, {2}", a, b, c); // 2, 3, 1
```

Stejně jako při prohazování dvou proměnných, i zde potřebujeme nějakou pomocnou proměnnou na odložení některé z hodnot. Je teoreticky jedno, hodnotu které z proměnných si do ní odložíme, ale nejsnáze a nejpřehledněji se kód sestaví, pokud tou odloženou proměnnou bude ta *putovní*, tj. ta, která se přesouvá ze začátku fronty na její konec. Tu si tedy takto odložíme (zkopírujeme do **p**), a jelikož ji máme *zálohovanou*, lze její (pouze její a žádné jiné) hodnotu přepsat tou, která se tam má přesunout. Tím máme zkopírovanou na své první místo tu, jež byla druhá v pořadí a můžeme tou následující přepsat tuto atd. Až na závěr, poté co hodnotu poslední proměnné zkopírujeme do proměnné předposlední, můžeme na konec vrátit hodnotu odloženou v pomocné proměnné.

Jako pomůcka pro správný zápis pořadí proměnných v kódu musí platit vztahy, které naznačují šipky na následujícím obrázku.

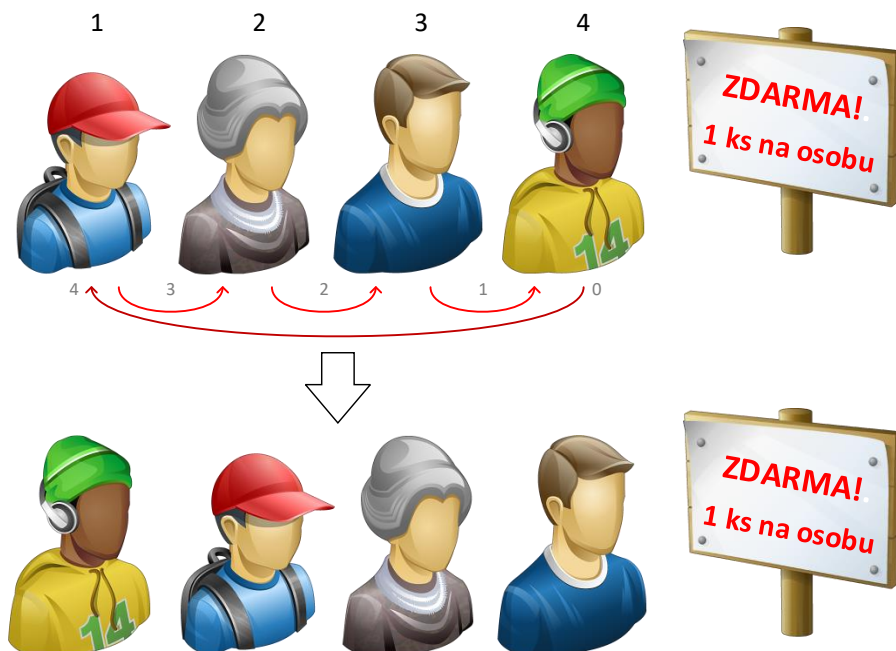


```
p = a;
a = b;
b = c;
c = p;
```



### 1.8.2 Cyklický posun vzad

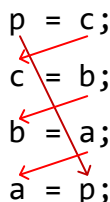
Cyklický posun vzad funguje stejně, jako ten vpřed. Rozdílný je pouze směr, ve kterém se hodnoty posouvají. V tomto případě jde o zpracování proměnných v opačném pořadí, než byly načteny (abecední řazení sestupné), u posloupností jde o sestupné řazení indexů jejích členů, a při výpisu hodnot za sebe do řádku by se posun dal nazvat posunem *doprava*. Celou situaci opět ilustruje následující obrázek fronty „na druhou stranu“.



Jak lze vypořádat z čísel operací, tak hlavním rozdílem je to, že se jednotlivé posuny musí vykonávat jednak opačným směrem, ale především v opačném pořadí. To je patrné i v následujícím jinak stejném kódu, tentokrát pro čtyři proměnné.

```
1: // Nastavení výchozích hodnot (lze načíst i klasickým způsobem)
2: int a = 1, b = 2, c = 3, d = 4;
3: // Cyklický posun vzad/doprava (1, 2, 3, 4 => 4, 1, 2, 3)
4: Console.WriteLine("{0}, {1}, {2}, {3}", a, b, c, d); //1, 2, 3, 4
5: int p = d; // Poslední (putovní) proměnná se odloží
6: d = c;
7: c = b;
8: b = a;
9: a = p;      // Odložená (putovní) proměnná se vrátí do té první
10: Console.WriteLine("{0}, {1}, {2}, {3}", a, b, c, d); //4, 1, 2, 3
```

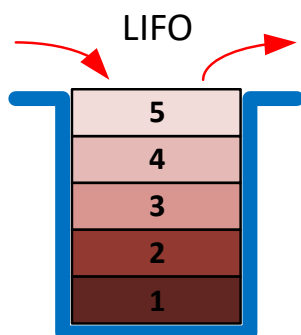
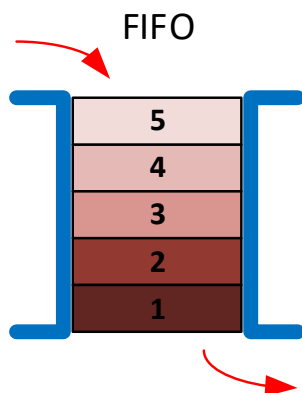
Putovní proměnnou je tentokrát ta poslední, takže do pomocné proměnné si zálohujeme tu a od toho se odvíjí i celý následný sled operací. Co se týče vztahů mezi proměnnými v kódu, tak ty jsou zcela totožné, jako při posunu vpřed, pouze názvy proměnných jsou seřazeny opačně.



```
p = c;
c = b;
b = a;
a = p;
```

### 1.8.3 Fronta a zásobník

Cyklické posuny by mohly evokovat podobnost mezi tzv. *frontou* či *zásobníkem*. Jelikož byly oba příklady demonstrovány na „zacyklené“ nekonečné *frontě*, tak podobnost s frontou tu samozřejmě je. Ta je však v tomto případě bez opakování (odbavené prvky se standardně nevracejí) a obvykle i s možností, že se do této fronty za běhu řadí další prvky. Takováto **fronta** se nazývá zkratkou **FIFO** (First In – First Out, tj. první dovnitř – první ven, nebo taky „kdo dřív přijde, ten dřív mele“). Představit se dá také jako tunel, kde nelze předjíždět. Tato forma zpracování prvků/osob/požadavků/operací je přirozenější a nezbytná je např. ve skladu potravin, kde je třeba zpracovávat potraviny postupně, tak jak byly naskladněny, aby tam žádné nezůstávaly příliš dlouho a nekazily se.



Druhou možností, jak určité prvky zpracovávat, je **LIFO** (Last In – First Out, čili kdo přišel poslední, odejde první). Jednoslovné české označení tohoto přístupu je **zásobník** a představíme-li si zásobník do pistole, tak ten přesně vystihuje, v jakém pořadí jsou z něho náboje vystřelovány (poslední vložený náboj bude vystřelen jako první). Představit si to můžeme také jako hromadu položek, na

které neustále vršíme další a když si nějakou potřebujeme vzít a zpracovat, tak nemáme jinou možnost než si vzít tu ze shora. Tento přístup by u potravin asi nebyl vhodný (ty dole by se brzo zkazily), ale např. u cihel, prken nebo košíků v obchodě je více praktický než pro odběr dolovat položku ze spodu hromady či druhého uzavřeného konce zásobníku.

## 1.9 Cykly

Cykly umožňují určitou část kódu vykonávat opakovaně, aniž by musela být zapsána vícekrát. Kód, který je určen k cyklickému opakování je obvykle vymezen začátkem a koncem (v C# složenými závorkami), mimo případu, kdy má být opakován jen jediný příkaz (tam závorky nejsou nutné). Těmto závorkám (či samostatnému příkazu) předchází příkaz cyklu, obvykle zahrnující podmínku, jež rozhoduje o pokračování či ukončení dalších opakování cyklu.

### 1.9.1 Cyklus s podmínkou na začátku

Základním typem cyklu je cyklus s podmínkou na začátku, též často zvaný **while** (česky *dokud*). Všechny ostatní typy cyklů, včetně **for** či cyklu s podmínkou na konci, lze tímto nahradit. Jeho syntaxe je přitom prostá: klíčové slovo **while**, za ním v kulatých závorkách podmínka, která dokud platí, tak se následující kód opakuje, jakmile platit přestane, cyklus končí a program pokračuje následujícím řádkem. Ověřování platnosti podmínky se provádí vždy před každým kolem cyklu, včetně toho prvního.

```
1: int i = 1;           // Deklarace a výchozí hodnota počítadla
2: while (i <= 10)      // Opakuj, dokud platí podmínka
3: {                   // Začátek cyklu (opakujícího se kódu)
4:     Console.WriteLine(i); // Vypíše aktuální hodnotu proměnné i
5:     i++;              // i++ je totéž jako i = i + 1;
6: }                   // Konec cyklu
```

Výše uvedený kód pod sebe vypíše do konzolového okna čísla od 1 do 10 (včetně). Dolní hranici lze změnit na ř. 1, horní v podmínce na ř. 2, krokování na ř. 5 a opakovaný příkaz či více příkazů pak na ř. 4.

Následující tři kódy dělají totéž a mají i naprosto stejný výstup, jen přistupují jiným způsobem ke zvyšování hodnoty v proměnné *i*.



```
1: int i = 1;
2: while (i <= 10)
3:     Console.WriteLine(i++);
```

Zvyšování proměnné **i** o +1 zde není jako samostatný příkaz, ale zvyšuje se v rámci jeho čtení (až po něm) pro vypsání do okna konzole (ř. 3). Díky tomu má nyní cyklus pouze jeden příkaz pro opakování a ušetří se tak další dva řádky pro závorky začátku a konce bloku cyklu.

```
1: int i = 0;
2: while (++i <= 10)
3:     Console.WriteLine(i);
```

Tento způsob má opět tentýž výstup, avšak navýšování proměnné **i** probíhá tentokrát již v rámci jeho testování v podmínce **while** (ř. 2), díky čemuž tuto proměnnou výkonný příkaz (ř. 3) používat vůbec nemusí, byť tomu tak v tomto případě je. Proto byla posunuta výchozí hodnota počítadla na 0.

A ještě jeden totéž vykonávající (o něco delší) příklad.

```
1: int i = 1;
2: while (true)
3: {
4:     if (i > 10)
5:         break;
6:     Console.WriteLine(i++);
7: }
```

Tento způsob není nejvhodnějším použitím cyklu **while**, ale funguje a dobře prezentuje efekt příkazu **break** (ř. 5). Ten totiž okamžitě ukončí běh cyklu (i nekonečného, jako by jinak byl tento), přeskočí i případné další příkazy po něm v rámci bloku cyklu následující (ř. 6) a pokračuje až za ním (za ř. 7).

Další příklad tentokrát produkuje jiný výstup, a to výpis pouze lichých čísel od 1 do 9.

```
1: int i = 0;
2: while (++i <= 10)
3: {
4:     if (i % 2 == 0)
5:         continue;
6:     Console.WriteLine(i);
7: }
```

Tento příklad opět není ukázkou optimálního řešení problému, ale slouží pro prezentaci příkazu **continue** (ř. 5). Ten podobně jako předchozí příkaz **break** přeskakuje část kódu, avšak tentokrát z cyklu nevyskočí zcela, ale pouze vynechá další zpracování aktuálního kola cyklu. V tomto případě to znamená, že pokud podmínka vyhodnotí, že je *i* sudé (ř. 4), provede příkaz **continue** (ř. 5), který přeskočí zbytek kódu v cyklu (ř. 6) a pokračuje s kolem dalším, opět od začátku, navýšením *i* a testováním podmínky cyklu (ř. 2).

### 1.9.2 Cyklus FOR

Cyklu **for** se obecně označuje jako „cyklus s předem známým počtem opakování“, protože standardně se používá tak, že je mu určena dolní mez (číslo na kterém začne), horní mez (číslo na kterém skončí) a krokování, tedy zvyšování proměnné o +1.<sup>5</sup> Tento případ ukazuje následující kód.

```
1: for (int i = 1; i <= 10; i++)
2:     Console.WriteLine(i);           // 1,2,3,4,5,6,7,8,9,10
```

---

<sup>5</sup> Tuto základní konstrukci kódu cyklu **for** lze ve Visual Studiu vygenerovat pomocí tzv. *snippet* tak, že napíšete „**for**“ a stisknete 2x klávesu tabulátoru.

Jak již bylo zmíněno dříve, definice tohoto cyklu se krom klíčového slova **for** skládá ze tří částí zapsaných v následné závorce a oddělených středníky. První část (`int i = 1`) je provedena pouze poprvé a obvykle se používá pro deklaraci a nastavení výchozí hodnoty proměnné počítadla. Druhá část (`i <= 10`) by měla obsahovat podmínku, která se stejně jako v cyklu **while** testuje před prvním i každým dalším kolem cyklu. Třetí část (`i++`) je určena pro příkaz, který je vykonáván před druhým a každým dalším kolem cyklu (ještě před testováním podmínky ve druhé části), tedy ideální místo pro navýšování počítadla.

Všechny tyto části se vyskytovaly i při použití cyklu **while**, pouze v té základní verzi byly rozepsány jako samostatné příkazy a zabíraly tak řádky navíc. Cyklus **for** je tedy z tohoto hlediska úspornější co do počtu řádků.

Veškeré části deklarace se přitom dají libovolně upravit, popř. i zcela vynechat (včetně podmínky, ta pak vychází vždy jako **true**). Pokud například v poslední části změníme `i++` na `i+=2`, bude se počítadlo v každém kole zvyšovat nikoli o +1 ale o +2 a v případě že `i` začíná na 1, bude počítadlo pracovat pouze s lichými čísly (1, 3, 5, 7, 9...).

Lze také použít snižování počítadla (např. `i--`) pro získání jeho sestupných hodnot, přičemž je ale nutné patřičně upravit výchozí hodnotu i podmínku (např. `i >= 1`), což ukazuje následující příklad.

```
1: for (int i = 10; i >= 1; i--)  
2:     Console.WriteLine(i);           // 10,9,8,7,6,5,4,3,2,1
```

Také v cyklu **for** fungují příkazy **break** a **continue**, stejně jako tomu bylo u cyklu **while**.

Existuje také cyklus **foreach**, který nemá žádné počítadlo a je určen pro procházení prvků nějaké posloupnosti (kap. 2.1) či seznamu (kap. 2.3). V každém kroku svého opakování tak postupně načítá jeden prvek z daného seznamu za druhým do zvláštní proměnné a jeho pod-kód s ním může něco vykonat. S prvky zpracovávaného seznamu však při jeho procházení manipulovat nelze.

```
1: foreach (var prvek in seznam) // Projde seznam prvků
2:     Console.WriteLine(prvek); // Vypíše prvky seznamu na řádky
```

### 1.9.3 Opakované zadávání hesla

Nyní si vyzkoušíme, jak cykly využít o něco praktičtěji. Následující kód například ukazuje, jak cyklus **while** blokuje pokračování v běhu programu, dokud není zadán určitý textový řetězec (správné heslo „123“).

```
1: // První zadání hesla
2: Console.Write("Zadejte heslo: ");
3: string heslo = Console.ReadLine();
4: // Ověření hesla a případné další pokusy na jeho zadání
5: while (heslo != "123")
6: {
7:     Console.Write("Neplatné heslo, zkuste to znovu: ");
8:     heslo = Console.ReadLine();
9: }
10: // Tajná sekce, dostupná pouze po zadání kódu
11: Console.WriteLine("Tajná informace...");
```

Program vyzve uživatele k zadání hesla (ř. 2–3) a jeho správnost testuje rovnou podmínka cyklu (ř. 5). Pokud (*dokud*) heslo není správné, je na tuto skutečnost uživatel upozorněn a vyzván k dalšímu pokusu (ř. 7–8). Těchto má v tomto případě neomezené množství, avšak nezadá-li správné heslo,



nemůže se nikdy z cyklu dostat ven (na ř. 11) a jedinou možností, jak program opustit bez hesla je jeho vypnutí (zavření konzolového okna křížkem, nebo ukončením aplikace).

Vhodnější variantou by jistě bylo, přidat možnost, aby se program dal ukončit zadáním určitého výrazu zadaného místo hesla (např. „konec“), anebo omezit počet pokusů na určitý počet, jak ukazuje následující příklad, jež je rozšířením toho předchozího.

```
1: // První zadání hesla
2: Console.Write("Zadejte heslo: ");
3: string heslo = Console.ReadLine();
4: int i = 1;    // Počítadlo pokusů
5: // Ověření hesla a případné další pokusy na jeho zadání
6: while (heslo != "123" && i < 3) // Max. 3 pokusy
7: {
8:     Console.Write("Neplatné heslo, zkuste to znovu: ");
9:     heslo = Console.ReadLine();
10:    i++;
11: }
12: // Zde buď bylo zadáno správné heslo, nebo došly pokusy
13: if (heslo == "123")
14: { // Tajná sekce, dostupná pouze po zadání kódu
15:     Console.WriteLine("Tajná informace...");
16: } else
17:     Console.WriteLine("Počet pokusů byl překročen...");
```

Zde je do programu přidáno počítadlo pokusů *i* (ř. 4 a 10) a rozšířena podmínka pro opuštění cyklu (ř. 6), zdali tento počet již nepřekročil povolené maximum, v tomto případě 3. Tentokrát je však nezbytné za cyklem ještě otestovat, jakým způsobem došlo k jeho opuštění (ř. 13), jestli zadáním správného hesla (ř. 15), nebo překročením maximálního počtu pokusů (ř. 17) a podle toho adekvátně reagovat.



### 1.9.4 Fibonacciho posloupnost

Jednou z dalších typických úloh, kde se cykly dají využít, je generování číselných řad a posloupností. Mezi ně patří i proslulá *Fibonacciho posloupnost*, která je definována následovně:  $a_0 = 0$ ,  $a_1 = 1$ ,  $a_i = a_{i-1} + a_{i-2}$ . Následující příklad ukazuje, jakým způsobem je možné tuto řadu vypsát, od nultého až po její  $n$ -tý prvek.

```
1: // Vyžádat si od uživatele hodnotu n
2: Console.Write("Po kolikátý člen Fibonacciho posl. (1-93): ");
3: int n = Convert.ToInt32(Console.ReadLine());
4: // Vypsání prvních dvou členů posl. (s idnexy 0 a 1)
5: ulong a1 = 0, a2 = 1;
6: Console.WriteLine("{0,2}. = {1,26:N0}", 0, a1);
7: Console.WriteLine("{0,2}. = {1,26:N0}", 1, a2);
8: // Vypsání zbytku posloupnosti (indexy 2 až n)
9: for (int i = 2; i <= n; i++)
10: {
11:     ulong a3 = a1 + a2; // Výpočet následujícího člena
12:     Console.WriteLine("{0,2}. = {1,26:N0}", i, a3); // Vypsání
13:     a1 = a2; // Posun předchozích členů, aby se v příštím...
14:     a2 = a3; // .. kole počítalo s těmi pravými
15: }
```

Nejprve uživatel zadá index posledního člena, po který až chce *Fibonacciho posloupnost* vypsát (ř. 2–3). Její první dva členy posloupnosti jsou definovány pevně (ř. 5) a proto jsou i vypsány zvlášť, ještě před samotným cyklem (ř. 6–7). Zbytek požadované části řady se pak již zpracuje a vypíše v cyklu **for** (ř. 9–15). Jelikož nultý a první člen byl již vypsán dříve, cyklus (ř. 9) začíná od druhého člena ( $i=2$ ), postupuje po jedné ( $i++$ ) a končí až na hodnotě  $n$  včetně ( $i \leq n$ ).

Pro výpis členů posloupnosti je použito formátování čísla prostřednictvím parametrů `{1,26:N0}`. Číslo se tedy naformátuje dle národního formátu (tj. mezery mezi řády tisíců) a **0** za **N** z výpisu vynechá veškeré desetiny. Hodnota **26** za čárkou za indexem parametru pak doplní výsledný textový řetězec mezerami zleva na celkovou šířku 26 znaků, což je délka největšího čísla, které bude schopen program vygenerovat. Díky tomu budou mít hodnoty jednotlivých členů ve výpisu zarovnané řády jednotek hezky pod sebou, jak ukazuje následující (zkrácená) ukázka tohoto výstupu.

Po kolikátý člen Fibonacciho posl. (1-93): 94

0. =	0
1. =	1
2. =	1
3. =	2
4. =	3
5. =	5
6. =	8
7. =	13
...	
44. =	701 408 733
45. =	1 134 903 170
46. =	1 836 311 903
...	
91. =	4 660 046 610 375 530 309
92. =	7 540 113 804 746 346 429
93. =	12 200 160 415 121 876 738
94. =	1 293 530 146 158 671 551

V programu byl použit typ **int** pro určení indexu maximálního člena (**n**, ř. 3) a pro počítadlo kroků cyklu (**i**, ř. 9). Pro hodnoty členů posloupnosti (**a1**, **a2** a **a3**, ř. 5 a 11) je však použit typ **ulong**, jež má rozsah hodnot, se kterými dokáže počítat od 0 do  $2^{64}$  (viz kap. 1.1.3). I ten má však své limity, které byly právě posledním členem posloupnosti (94.) překročeny, hodnota tzv.

*přetekla přes své maximum a začala se znovu načítat od 0. Proto je 94. člen ve výpisu menší, než 93., a proto jeho hodnota již neodpovídá skutečné hodnotě 94. člena *Fibonacciho posloupnosti*. Proto je nezbytné s limity jednotlivých datových typů neustále počítat a vhodně je volit již během návrhu programu.*

K dispozici je nicméně ještě datový typ (struktura) **BigInteger**, který se do paměti neukládá běžným způsobem. Z tohoto důvodu práce s ním není natolik optimalizována jako u klasických datových typů s vymezeným rozsahem. **BigInteger** však není rozsahově, jak do záporných, tak do kladných hodnot, nijak omezen, díky čemuž dokáže zpracovávat libovolně velká celá čísla, pokud nepřesáhnou dostupný rozsah paměti. Pokud bychom tedy nahradili typ **ulong** za **BigInteger** (na ř. 5 a 11), tak např. přesně vypočítat a vypsat 10 000 člen *Fibonacciho posloupnosti* s 2 090 ciframi (33.645E+2088) není problém, aniž by došlo k výraznému nárůstu paměti požadované aplikací.

## 2 Vícehodnotové proměnné

*Pole* jsou v paměti uchovávané sady hodnot stejného datového typu, které jsou z programátorova hlediska uchovávány pod názvem jedné proměnné, která je ovšem navíc doplněna indexem či více indexy. Záleží na tom, kolik rozměrů určíme, že má pole mít. V následujících kapitolách si ukážeme práci s jednorozměrným a dvourozměrným polem, avšak rozměrů lze analogickým způsobem polím přiřadit i více.

### 2.1 Posloupnosti (1D pole)

Jednorozměrná pole lze také nazývat posloupnostmi nebo vektory. Proměnná odkazující na pole musí být při přístupu k jejich hodnotám doplněna o index dané hodnoty, který se zapisuje za tuto proměnnou do hranatých závorek. Např. **pole[1]** odkazuje na posloupnost hodnot **pole** a v nich na hodnotu s indexem **1**.

Zde je ovšem rovnou nezbytné poznamenat, že index **1** není prvním indexem posloupnosti, protože tím je index **0**. Takže v případě že posloupnost má **N** členů, jsou jejich indexy **0, 1, ..., N-2, N-1**. Proto cykly zpracovávající celou posloupnost začínají s počítadlem indexu **i** na **i=0** a končí podmínkou **i<N** (*menší, nikoli menší nebo rovno*).

#### 2.1.1 Základní operace s posloupnostmi

Základním příkazem, který je třeba před používáním proměnné odkazující se na posloupnost provést, je její deklarace.

```
1: // Vytvoření (deklarace) jednorozměrného pole celých čísel
2: int[] a = new int[10]; // indexy 0-9 = 10 hodnot (členů)
3: int[] b = new int[] { 5, 3, 9 }; // 3 členy s danými hodnotami
```

Příklad ukazuje dva způsoby deklarace posloupností (**a** a **b**). První z nich (ř. 2) vytvoří posloupnost celých čísel (**int**) **a** s deseti členy (jejich indexy jsou tedy **0–9**, počet členů může být samozřejmě definován i proměnnou celočíselného typu). Hodnoty jim zatím nebyly nastaveny, ale jelikož datový typ **int** musí mít hodnotu vždy (jde o strukturu nikoli třídu), je všem členům nastavena výchozí hodnota (**default**<sup>6</sup>) tohoto typu, což je v případě číselných typů **0**.

Druhý typ deklarace (ř. 3) je méně častý, a umožňuje hodnoty posloupnosti rovnou nastavit jejich vypsáním za ně, do složených závorek. V tomto případě není třeba uvádět délku posloupnosti, neboť ta je odvozena od počtu členů, které jsou posloupnosti nastavovány (tj. 3, indexy 0-2). V obou případech může být jako typ proměnné uvedeno místo **int[]** klíčové slovo **var**.

Jakmile je posloupnost deklarována, lze pracovat s jejími hodnotami. Uvažovat budeme spíše první variantu deklarace, takže si hodnoty členů nejprve nastavíme (hned třemi různými způsoby).

```
1: // Načtení hodnot do pole od uživatele
2: for (int i = 0; i < a.Length; i++)
3:     a[i] = Convert.ToInt32(Console.ReadLine());
4:
5: // Nastavení hodnot řadou čísel od 1 do N
6: for (int i = 0; i < a.Length; i++)
7:     a[i] = i + 1;
8:
9: // Načtení náhodných hodnot (v rozsahu 0-9)
10: for (int i = 0; i < a.Length; i++)
11:     a[i] = random.Next(10); // random je třeba deklarovat zvlášť
```

---

<sup>6</sup> Výchozí hodnotu pro daný datový typ získáme přes klíčové slovo/funkci **default**, kde do závorek za něj uvedeme název tohoto typu (struktury). Např. `int x = default(int);`

Všechny tři způsoby nastavování hodnot členům posloupnosti používají tentýž cyklus. Ten začíná na hodnotě **0** (index prvního člena posloupnosti) a jeho zakončovací podmínka je `i < a.Length`. Proměnná **a** se odkazuje na dříve deklarovanou posloupnost, a kromě možnosti přistupovat k jejím členům přes index, disponuje i několika vlastnostmi a metodami pro usnadnění práce s posloupnostmi. Vlastnost **Length** udává počet členů v posloupnosti, tedy hodnotu značenou někdy jako **N** a v předchozím příkladě při deklaraci nastavenou na **10**. Cyklus tedy v proměnné **i** postupně vystřídá všechny celočíselné hodnoty od **0** do **9**.

První způsob nastavení členů (ř. 2–3) načítá tyto hodnoty od uživatele z příkazové řádky. Ten tedy musí 10x zadat číslo a potvrdit jej stiskem klávesy Enter.

Druhý způsob (ř. 6–7) členům nastavuje hodnoty shodné s jejich indexem navýšeným o 1. V posloupnosti tedy na indexech 0–9 budou hodnoty 1–10.

Třetí způsob (ř. 10–11) členům nastaví celočíselné náhodné hodnoty v rozsahu 0–9 („nedosažitelné“ maximum pro generátor je nastaveno na 10). Abychom ovšem generátor náhodných hodnot mohli použít, je třeba jej (proměnnou **random**) nejprve deklarovat, a to ideálně jen na jednom místě pro celý program (aby se generovaly neustále další náhodné hodnoty).

```
1: class Program
2: {
3:     static Random random = new Random(); // Deklarace generátoru
4:     static void Main(string[] args) { ... } // Kód programu
5: }
```

V našem případě můžeme generátor v proměnné **random** deklarovat (ř. 3) jako statickou proměnnou v rámci hlavní třídy **Program**.

Hodnoty v posloupnosti máme nastavené, ať již jakýmkoli způsobem, a chceme-li je také zobrazit, pak je třeba je vypsat.

```
1: // Výpis posloupnosti - pod sebe
2: for (int i = 0; i < a.Length; i++)
3:     Console.WriteLine("{0}. {1}", i, a[i]);
```

Tento způsob vypíše hodnoty pod sebe, tj. každý jednotlivý člen na vlastní řádek, přičemž před hodnotu člena vypíše i jeho index (oddělený tečkou, např. 2. 8, tj. člen s indexem 2 má hodnotu 8.

Jelikož výpis posloupnosti může být v celku častou úlohou, vytvoříme si pro jeho výpis rovnou vlastní metodu.

```
1: /// <summary>Výpis posloupnosti - do řádku</summary>
2: static void VypisPosloupnost(int[] a)
3: {
4:     for (int i = 0; i < a.Length; i++)
5:         Console.Write("{0}{1}", a[i], i < a.Length-1 ? ", " : "");
6:     Console.WriteLine();
7: }
```

Metoda **VypisPosloupnost** tedy dostane na vstupu referenci na libovolnou posloupnost celých čísel a vypíše hodnoty jejích členů tentokrát do řádku za/vedle sebe, bez indexů a oddělené čárkami (např. 5, 3, 9). Výpis čárky je zde řešen také pomocí parametru textového řetězce (**{1}**), a to z toho důvodu, že čárka bude vypsána pouze v případech, že se nejedná o poslední vypisovaný člen (**i < a.Length-1**), za kterým tato čárka (na konci řádku) již nebude (místo ní se vypíše prázdný textový řetězec **""**). Tuto funkčnost řeší podmíněný operátor **?:** (viz kap. 1.1.2).



### 2.1.2 Výpočty v posloupnosti

Posloupnost je tedy vytvořená a naplněná hodnotami, které už si dokonce můžeme vypsát do konzole. Jak již bylo patrné z těchto přípravných programů, hlavní výhodou polí oproti jednoduchým proměnným je možnost jejich zpracování v cyklech. Proměnná má totiž svůj název, na který se musíme v kódu přímo odkazovat, avšak u polí je zde ještě index, který můžeme nahradit proměnnou a tu cyklicky měnit.

Dále si ukážeme, jak hodnoty v posloupnostech zpracovávat cyklicky na základních úkonech a výpočtech. Jednou z nejjednodušších úloh, kterou lze s posloupností čísel provést je vypočítat její součet a z něho pak průměr.

```
1: // Součet a aritmetický průměr členů posloupnosti
2: int soucet = 0;
3: for (int i = 0; i < a.Length; i++)
4:     soucet += a[i];
5: Console.WriteLine("součet = {0}", soucet);
6: Console.WriteLine("průměr = {0:N3}", soucet / (double)a.Length);
```

Pro výpočet součtu tedy založíme a vynulujeme počítadlo **soucet** (ř. 2), cyklem projdeme všechny členy posloupnosti (ř. 3) a hodnotu každého člena do tohoto součtu přidáme (ř. 4). Následně stačí součet již jen vypsát (ř. 5), popř. spočítat a rovnou i vypsát průměr z těchto hodnot (ř. 6), jako podíl součtu hodnot členů a jejich počtu.

Místo **for** cyklu by v tomto případě bylo možné použít i cyklus **foreach**, jelikož indexy prvků v těle cyklu tentokrát nepotřebujeme.

```
3: foreach (int x in a)
4:     soucet += x;
```

Pokud bychom potřebovali do výpočtu započítat pouze členy s určitou vlastností, např. sudou hodnotou, pak by kód mohl vypadat následovně.



```
1: // Počet sudých členů bez 0
2: int pocet = 0;
3: for (int i = 0; i < a.Length; i++)
4:     if (a[i] % 2 == 0 && a[i] != 0)
5:         pocet++;
6: Console.WriteLine("Počet sudých hodnot (bez 0) je {0}", pocet);
```

Tentokrát jsme tedy počítali pouze počet sudých hodnot (do nichž jsme nezapočítávali ani hodnotu 0). Kód je podobný předchozímu příkladu, byť místo součtu zvyšujeme pouze počet o +1 (ř. 5), ovšem hlavní rozdíl je v podmínce omezující započítání pouze hodnot s danou vlastností (ř. 4). Pokud bychom chtěli např. znát průměr pouze ze sudých hodnot, muselo by pod tuto podmínku spadat jak zvyšování počtu, tak i načítání součtu. To už by byly ovšem dva příkazy a musely by být přidány i složené závorky vymezující blok (rozšířenou platnost) této podmínky (tj. otevírací mezi řádky 4 a 5, zavírací mezi řádky 5 a 6).

### 2.1.3 Extrémy

Pokud budeme v posloupnosti hledat nějaký extrém, tj. minimum či maximum, resp. nejmenší či největší hodnotu členů v poli, můžeme postupovat analogicky jako při hledání nejvyššího sněhuláka (viz kap. 1.6.3). Postup hledání minima ukazuje následující kód.

```
1: // Nalezení minimální hodnoty
2: int min = a[0]; // Výchozí minimum bude hodna 1. člena (index 0)
3: for (int i = 1; i < a.Length; i++) // Projít zbytek (od 1) posl.
4:     if (a[i] < min) // Je hodnota člena menší než dosavadní min?
5:         min = a[i]; // Pokud ano, novým min. budiž tato hodnota
6: Console.WriteLine("minimum = {0}", min); // Vypsát výsledek
```



Minimum je sice konkrétní číslo, ale jeho hodnoty může nabývat vícero členů posloupnosti, konkrétně 1–N. Pokud jsme již minimum našli, můžeme počet jeho výskytů určit následujícím kódem.

```
1: // Počet minim
2: int pocetMin = 0;
3: for (int i = 0; i < a.Length; i++)
4:     if (a[i] == min)
5:         pocetMin++;
6: Console.WriteLine("počet minim je {0}", pocetMin);
```

V případě, že minimum dosud neznáme a budeme jej teprve hledat, můžeme kromě jeho hodnoty rovnou určit i počet jeho výskytů v rámci jediné cyklu. Takový kód by pak mohl vypadat třeba následovně.

```
1: // Nalezení minimální hodnoty a určení počtu jejích výskytů
2: int min = a[0]; // Výchozí minimum
3: int pocetMin = 1; // Hodnotu vých. minima už jsme jednou našli
4: for (int i = 1; i < a.Length; i++) // Prohledat zbytek posl.
5:     if (a[i] < min) // Nalezeno nové minimum
6:     {
7:         min = a[i]; // Uložit si jeho hodnotu
8:         pocetMin = 1; // Tuto hodnotu jsme našli poprvé
9:     }
10:    else if (a[i] == min) // Člen je roven starému minimu
11:        pocetMin++; // Započítat jeho výskyt
12: Console.WriteLine("minimum {0} tam je {1}x", min, pocetMin);
```

Pokud by nás ovšem zajímala pozice, resp. index, tohoto minima, lze celý proces řešit i právě výhradně za pomoci tohoto indexu, bez nutnosti dalších proměnných.



```
1: // Index prvního minima
2: int iMin = 0; // Výchozí minimum bude 1. člen (index 0)
3: for (int i = 1; i < a.Length; i++) // Projít zbytek (od 1) posl.
4:     if (a[i] < a[iMin]) // Je člen menší než dosavadní min?
5:         iMin = i; // Pokud ano, nové min je člen na tomto indexu
6: Console.WriteLine("minimum = {0} a poprvé je na indexu {1}",
                    a[iMin], iMin); // Vypsát výsledek
```

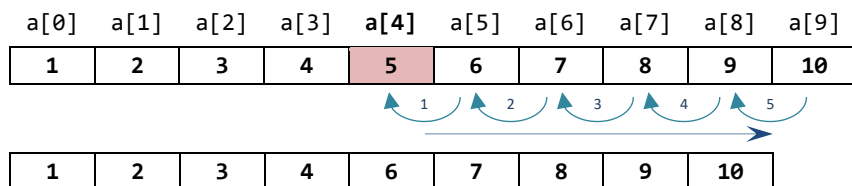
Program si tedy uchovává pouze index člena (ř. 2 a 5) s minimální hodnotou (popř. zprvu kandidáta na něj) a potřebuje-li znát jeho hodnotu, kdykoli ji z této posloupnosti načte právě přes tento index (ř. 4 a 6).

Jelikož ovšem v posloupnosti může být členů s hodnotou minima více, je třeba se také rozhodnout, index kterého z nich nás zajímá. Tento kód hledá index minima prvního. Pokud bychom chtěli znát index minima posledního, stačilo by v tomto kódu na řádku 4 změnit znak porovnání z `<` na `<=`.

A v případě, že by nás místo minima zajímalo maximum, stačí v každém z těchto příkladů při porovnávání otočit „zobáček“ z `<` na `>`.

## 2.1.4 Změny struktury posloupnosti

Členové posloupnosti jsou v ní uloženy v pevném pořadí pod svými indexy. Jejich hodnoty lze číst i nastavovat, ale co když je třeba do posloupnosti člen vložit, nebo jej odstranit? To totiž ovlivní pozice i členů následujících.

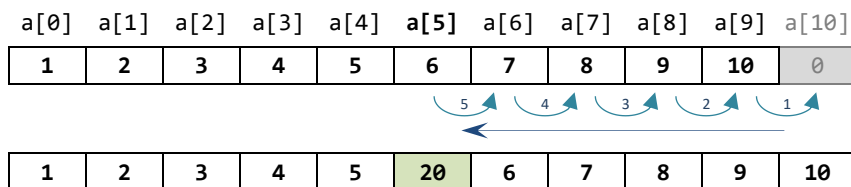


Při odebrání člena posloupnosti (na obr. člen s indexem 4 a hodnotu 5) stačí postupně (od pozice tohoto odebíraného člena ke konci) posouvat (kopírovat) hodnoty členů následujících na jim předchozí. Závěrečným krokem je pak zkrácení délky posloupnosti o posledního člena.

```
1: // Odebrání člena posloupnosti
2: Console.WriteLine("Zadejte index člena pro odebrání: ");
3: int k = Convert.ToInt32(Console.ReadLine());
4: VypisPosloupnost(a);           // Vypsání posloupnosti PŘED
5: for (int i = k; i < a.Length - 1; i++)
6:     a[i] = a[i + 1];           // Posun hodnot vpřed
7: Array.Resize(ref a, a.Length - 1); // Změna délky pole o -1
8: VypisPosloupnost(a);           // Vypsání posloupnosti PO
```

Změnu délky posloupnosti umí zařídit statická metoda **Resize** třídy **Array**. Té se s modifikátorem **ref**<sup>7</sup> předá měněná posloupnost a nová délka, na kterou ji chceme velikost změnit (v tomto případě zmenšit).

Při vkládání nového člena do posloupnosti je pak postup ve všech ohledech opačný.



<sup>7</sup> Parametry metod s modifikátorem **ref** metodě nejen že propůjčí svou hodnotu, ale pokud ji metoda ve svém kódu změní, změna se promítne také ven, do proměnné v tomto parametru (**a**). Nové zkrácené pole tedy bude přiřazeno zpět do proměnné **a**.

Nejprve se rozšíří pole o jeden nový (zatím prázdný, resp. nulový) člen na konci, od něho (od konce k pozici člena nově vkládaného) se budou posouvat (kopírovat) hodnoty členů vždy o jeden index výše a na závěr se hodnota nového člena vloží na zvolenou pozici.

```
1: // Vložení nového člena do posloupnosti
2: Console.Write("Zadejte index člena pro vložení: ");
3: int k = Convert.ToInt32(Console.ReadLine());
4: Console.Write("Zadejte hodnotu nového člena pro vložení: ");
5: int h = Convert.ToInt32(Console.ReadLine());
6: VypisPosloupnost(a); // Vypsání posloupnosti PŘED
7: Array.Resize(ref a, a.Length + 1); // Změna délky pole o +1
8: for (int i = a.Length - 1; i > k; i--)
9:     a[i] = a[i - 1]; // Posun hodnot vzad
10: a[k] = h; // Dosazení h na vniklé místo
11: VypisPosloupnost(a); // Vypsání posloupnosti PO
```

## 2.2 Matice (2D pole)

S maticemi neboli dvourozměrnými poli, se ještě důkladněji prakticky seznámíme v kapitole 3, proto si zde ukážeme jen úplné základy.

### 2.2.1 Deklarace dvourozměrného pole

Deklarovat dvourozměrné pole lze už v základu různými způsoby.

```
1: // Vytvoření (deklarace) dvourozměrného pole celých čísel
2: int[,] a = new int[4, 5]; // 4 řádky, 5 sloupců = 20 členů
3: // Rozměry zadány proměnnými (m = počet řádků, n = sloupců)
4: int m = 4, n = 5;
5: int[,] b = new int[m, n]; // indexy řádků 0-3, sloupců 0-4
6: // Hodnoty (a tím i rozměry) zadány přímo v rámci deklarace pole
7: var c = new int[,] { { 1, 2, 3, 4, 5 },
8:                     { 6, 7, 8, 9, 10 },
9:                     { 11, 12, 13, 14, 15 },
10:                    { 16, 17, 18, 19, 20 } };
```

Jak je z ukázky patrné, tak to, že se na rozdíl od předchozí kapitoly jedná o pole dvourozměrné, naznačíme při deklaraci proměnné tak, že do hranatých závorek za datový typ členů matice přidáme čárku<sup>8</sup>. Ta určuje, že tam tentokrát budou indexy dva, touto čárkou oddělené. I zde lze ve všech případech nahradit přímý název datového typu proměnné při jeho deklaraci klíčovým slovem **var**, jak je použito u matice **c**. Hodnoty členů matic **a** a **b** budou opět defaultní (0) a v matici **c** budou hodnoty zadané při deklaraci.

V ukázce je řečeno, že první zadaný rozměr (4) určuje počet řádků a druhý (5) počet sloupců. Toto je varianta standardně používaná u matematických

---

<sup>8</sup> Pro více rozměrná pole bychom pak pokračovali analogicky, tj. u tří rozměrného (tří dimenzionálního, též 3D) pole by v deklaraci byly dvě čárky (`int[, , ]`) oddělující tři čísla (`int[1,2,3]`), u 4D pole 3 čárky, atd.

matic, kde první index, obvykle značený proměnnou **i** identifikuje řádek, jejichž počet bývá značen proměnnou **m**, a index druhý značený proměnnou **j** identifikuje sloupec (resp. již konkrétního člena na řádku **i**), jejichž počet bývá značen proměnnou **n**. Rozdíl oproti matematické verzi značení matice je však ten, že indexy jsou zde počítány od nuly (v mat. od 1), takže index posledního řádku není tentýž jako jejich počet **m**, ale **m-1** a index posledního sloupce je **n-1**.

$$\begin{pmatrix} a_{0,0} & a_{0,1} & \dots & a_{0,j} & \dots & a_{0,n-1} \\ a_{1,0} & a_{1,1} & \dots & a_{1,j} & \dots & a_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ a_{i,0} & a_{i,1} & \dots & a_{i,j} & \dots & a_{i,n-1} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ a_{m-1,0} & a_{m-1,1} & \dots & a_{m-1,j} & \dots & a_{m-1,n-1} \end{pmatrix}$$

Tohoto matematického značení, byť s indexací od 0, se budeme držet i ve všech následujících příkladech. Nicméně programátorovi samozřejmě nic nebrání si indexy prohodit a jako první indexovat sloupec a řádek až jako druhý (místo **[i,j]** uvažovat **[x,y]**). Záleží pak pouze na tom, jak bude s maticí pracovat a jak bude její hodnoty program interpretovat.

## 2.2.2 Naplnění matice

Máme-li proměnnou s maticí deklarovanou, můžeme jejím členům nastavit i nějaké nedefaultní hodnoty. Ukážeme si opět hned tři varianty (ř. 5–7).

```
1: // Naplnění matice (třemi různými způsoby)
2: for (int i = 0; i < a.GetLength(0); i++)
3:     for (int j = 0; j < a.GetLength(1); j++)
4:     {
5:         a[i, j] = j + i * a.GetLength(1) + 1; // po řádcích 1-20
6:         a[i, j] = i + j * a.GetLength(0) + 1; // po sloupcích
7:         a[i, j] = random.Next(10);           // náhodně 0-9
8:     }
```



Proces procházení celé matice je vždy stejný (ř. 2 a 3), varianty nastavování hodnot se tedy liší pouze uvnitř cyklů (ř. 5, 6 a 7), přičemž z těchto příkazů použijeme pouze jeden (pak je možné vynechat i závorky na ř. 4 a 8). První z nich (ř. 5) nastavuje členům matice hodnoty od 1 do 20 popořadě za sebou po řádcích, stejně jako byly v předchozí ukázce nastaveny hodnoty členů matice **c**. Druhá varianta (ř. 6) nastavuje hodnoty podobně ale po sloupcích (čili v prvním řádku je 1, 5, 9, 13, 17, ve druhém 2, 6, 10 atd.). Třetí varianta (ř. 7) pak generuje náhodná celá jednociferná čísla (v rozsahu 0–9), přičemž v tomto případě je nezbytné proměnnou **random** předem deklarovat, stejně jako tomu bylo v případě plnění posloupnosti náhodnými čísly.

Abychom minimalizovali redundantní proměnné, místo **m** a **n** budeme informaci o počtu řádků a sloupců matice opět čerpat přímo z ní. Vlastnost **Length** ovšem i tentokrát udává celkový počet členů, což je v tomto případě **m\*n** (4\*5, tedy 20). Tato informace je sice důležitá, ale počet řádků ani sloupců z ní nezjistíme. Za tímto účelem je zde pro vícerozměrná pole metoda **GetLength** (viz ř. 2 a 3), která jako vstupní parametr požaduje číslo (index) dimenze, čili 0 je 1. dimenze/index (**m**), neboli v našem případě řádky, a 1 je 2. dimenze/index (**n**), neboli sloupce.

### 2.2.3 Výpis matice

Abychom se mohli přesvědčit, jaké hodnoty v matici nakonec jsou, tak následující kód ukazuje metodu **VypisMatici**, která hodnoty členů matice zadané přes vstupní parametr vypíše do konzole. Tentokrát ponechá oddělovací čárku i za posledním členem každého řádku včetně posledního a za výpisem matice vynechá prázdný řádek.



```
1: /// <summary>Výpis matice</summary>
2: static void VypisMatici(int[, ] a)
3: {
4:     for (int i = 0; i < a.GetLength(0); i++)    // řádky
5:     {
6:         for (int j = 0; j < a.GetLength(1); j++) // sloupce
7:             Console.Write("{0}, ", a[i, j]);    // vypsát člena
8:             Console.WriteLine();                // zalomit řádek
9:     }
10:    Console.WriteLine();    // vynechat řádek pod maticí
11: }
```

Jak je vidět z předchozích dvou ukázek, tak týmiž dvěma cykly (pro *i* a pro *j*) se matice zpracovává v podstatě pokaždé. Pokud bychom pořadí těchto dvou cyklů prohodili (nejprve pro *j* a až pak pro *i*), výsledek by byl stejný, ale změnilo by se pořadí zpracování jednotlivých členů. Místo po řádcích (tak jak čteme a v tomto případě i do konzole vypisujeme) by program postupoval po sloupcích (první do shora dolů, pak druhý atd.).

#### 2.2.4 Výpočty v maticích

Jako poslední ukázkou práce s maticemi typu **(m, n)** uveďme opět klasický případ pro výpočet součtu hodnot všech členů matice a na základě toho i určení jejich aritmetického průměru.

```
1: // Součet a aritmetický průměr členů matice
2: int soucet = 0;
3: for (int i = 0; i < a.GetLength(0); i++)
4:     for (int j = 0; j < a.GetLength(1); j++)
5:         soucet += a[i, j];
6: Console.WriteLine("Součet členů matice je {0}", soucet);
7: Console.WriteLine("Průměr členů matice je {0}",
    soucet / (double)a.Length);
```

## 2.2.5 Čtvercové matice

Na závěr ještě poznamenejme, že toto byly obecné „obdélníkové“ matice, tzv. typu  $(m, n)$ , avšak někdy se rozlišují ještě matice čtvercové typu  $(m, m)$ , které jsou speciálním podtypem těch obdélníkových. Jak je z tohoto značení patrné, tak čtvercové matice mají stejný počet řádků jako sloupců ( $m$ ), díky čemuž u nich lze navíc definovat tzv. hlavní diagonálu (členy jejichž indexy se rovnají:  $i==j$ , tj. z levého horního rohu do pravého dolního, v ukázce zelená) a diagonálu vedlejší (tu protilehlou,  $j=m-i-1$ , z pravého horního rohu do levého dolního, v ukázce červená).

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

5 x 5

	0	1	2	3
0	1	0	0	2
1	0	1	2	0
2	0	2	1	0
3	2	0	0	1

4 x 4

Obě diagonály jsou v podstatě jednorozměrným útvarům, byť ve dvourozměrném prostoru, takže je lze zpracovat pouze jediným cyklem. Kód, který pro  $m=5$  (popř. i pro  $m=4$ ) vytvoří předchozí ukázkový výstup tak může vypadat následovně.

```

1: // Deklarace čtvercové matice
2: int m = 5; // Popř. ve druhé ukázce m = 4
3: int[,] a = new int[m, m]; // Matice celých čísel, vých. hodn. 0
4: // Hlavní diagonála - přičíst +1
5: for (int i = 0; i < m; i++)
6:     a[i, i] += 1;
7: // Vedlejší diagonála - přičíst +2
8: for (int i = 0; i < m; i++)
9:     a[i, m-i-1] += 2;
```

## 2.3 Seznamy

Kromě polí, na kterých jsme si ukázali základní algoritmické postupy, v C# existuje i celá řada dalších možností, jak uchovávat „více hodnot v jedné proměnné“. Obecně by se daly nazývat seznamy a jejich členy lze označovat za prvky či položky. Nejčastějším typem klasického seznamu je **List**.

```
1: List<int> a = new List<int>(); // Deklarace seznamu celých čísel
```

Při vytváření nové instance seznamu **List** se ve špičatých závorkách definuje tzv. generický datový typ, který určuje, jakého typu budou všechny prvky tohoto seznamu. V ukázce je pro ně použit opět celočíselný typ **int**. Délku seznamu není třeba určovat předem už v okamžiku jeho deklarace, protože do seznamu, na rozdíl od posloupnosti lze prvky libovolně přidávat (**Add**), vkládat (**Insert**) nebo je odebírat (**Remove**) i kdykoli později.

```
1: // Naplnění deseti náhodnými hodnotami
2: for (int i = 0; i < 10; i++)
3:     a.Add(random.Next(10)); // Přidá náhodné č. na konec seznamu
4: // Další základní změny v prvcích seznamu
5: a.Insert(5, 10); // Vloží 10 na index 5 (prvky za se posunou)
6: a.Remove(10);    // Odstraní první prvek s hodnotu 10
7: a.RemoveAt(5);   // Odstraní prvek na indexu 5
```

Mezi 1D poli (posloupnostmi) a seznamy lze celkem snadno převádět pomocí vestavěných funkcí **ToArray** a **ToList**.

```
1: List<int> seznam = new List<int>(); // Nový seznam
2: int[] pole = seznam.ToArray();      // Pole ze seznamu
3: seznam = pole.ToList();             // Seznam z pole
```

Převod mezi seznamem a polem je ovšem netriviální operace a na pozadí těchto funkcí je algoritmus, který bychom bez těchto funkcí museli napsat sami (vytvořit nový seznam/pole příslušné délky a zkopírovat do něho všechny položky ze zdrojového pole/seznamu), takže není vhodné jich využívat zbytečně často, zvláště pak u větších polí/seznamů.

Seznam lze vypsat různými způsoby. Z hlediska konzolového výstupu lze v základu prvky seznamu vypsat buď do sloupce (každý prvek na nový řádek), nebo do řádku (prvky za sebou oddělené čárkou). Dvě varianty výpisu prvků do řádku, cyklem **for** (ř. 2–4) a cyklem **foreach** (ř. 6–8), ukazuje následující kód.

```
1: // Vypsání seznamu do řádku cyklem for
2: for (int i = 0; i < a.Count; i++)
3:     Console.Write("{0}, ", a[i]);
4: Console.WriteLine(); // Zalomení řádku
5: // Vypsání seznamu do řádku cyklem foreach
6: foreach (int prvek in a)
7:     Console.Write("{0}, ", prvek);
8: Console.WriteLine(); // Zalomení řádku
```

Místo **Length** (délka) se počet prvků seznamu dozvíme z vlastnosti **Count** (počet, viz ř. 2). Prvky seznamu lze také indexovat pomocí hranatých závorek (viz ř. 3). Pokud index nepotřebujeme při daném procházení prvků znát, lze i zde použít cyklus **foreach** (ř. 6–7).

Pro výpis seznamu můžeme vytvořit také metodu, kterou pak budeme moci kdykoli zavolat (**Vypis(a);**). V následující ukázce je kód takovéto metody. Tentokrát je však metoda napsána maximálně univerzálně, tj. aby zvládla výpis nejen seznamu typu **List**, ale i pole, kolekce atd. přičemž typ prvků seznamu může být také libovolný (**int**, **string**, **double**, **bool**, ...).



```
1: /// <summary>Metoda, která vypíše většinu seznamů</summary>
2: static void Vypis<T>(IEnumerable<T> a)
3: {
4:     foreach (T x in a)
5:         Console.Write("{0}, ", x);
6:     Console.WriteLine();
7: }
```

## 2.4 Slovníky

Slovník (**Dictionary**) je poněkud odlišným typem seznamu oproti předchozím případům. Přirovnat by se dal snad ke dvourozměrnému poli (matici) se dvěma sloupci a vícero řádky, přičemž v prvním sloupci by byl tzv. klíč (**key**, hodnota jedinečná ve všech řádcích) a ve sloupci druhém tzv. hodnota (**value**). Datové typy v těchto dvou sloupcích by přitom mohly být různé.

Následující příklad ukazuje dva způsoby, jak lze slovník vytvořit, přičemž u druhého z nich je předvedeno i několik způsobů, jak do něho přidávat položky (páry klíč + hodnota).

```
1: // Vytvoření prázdného slovníku (klíč je číslo, hodnota je text)
2: var ciselnik = new Dictionary<int, string>();
3: // Vytvoření nového slovníku se dvěma položkami
4: var slovník = new Dictionary<string, string>() {
5:     { "car", "auto" },
6:     { "air", "vzduch" },
7: };
8: // Přidávání položek do slovníku
9: slovník.Add("blue", "modrá");
10: slovník["mirror"] = "zrcadlo";
11: slovník["car"] = "automobil"; // Nahradí hodnotu u položky "car"
```

První příkaz (ř. 2) založí nový prázdný slovník, v němž bude datový typ klíče celé číslo (**int**) a datový typ hodnot textový řetězec (**string**). Druhý příkaz (ř. 4–7) vytvoří také nový slovník, ovšem klíč i hodnota budou typu **string** a zároveň do tohoto slovníku rovnou vloží dvě položky, páry anglického a českého slovíčka, přičemž to anglické je klíčem a české hodnotou.

Další tři příkazy (ř. 9–11) ukazují, jak lze do vytvořeného slovníku přidávat položky. První (ř. 9) přidává pár klíč/hodnota pomocí metody **Add**, další dva pak ve slovníku „indexují“ položky přes jejich klíč (klíč je zapsán v hranatých závorkách podobně jako index posloupnosti). Pokud daný klíč (*mirror*) ve slovníku zatím není (ř. 10) je do něho přidán s nastavovanou hodnotu (*zrcadlo*). V případě, že se položka s tímto klíčem (*car*) již ve slovníku nachází, je pouze přepsána její hodnota (*auto* z ř. 5 na *automobil* na ř. 11). Pokud by však klíč již ve slovníku existoval a položku jsme přidávali metodou **Add**, došlo by k vyvolání výjimky (zaseknutí programu).

Jak ze slovníku získat hodnotu pro určitý klíč (existuje-li) ukazuje další kód.

```
1: // Kontrola, je-li klíč "car" ve slovníku, pokud ano vypíše se
2: if (slovník.ContainsKey("car"))
3:     Console.WriteLine("{0} = {1}", "car", slovník["car"]);
```

V tomto případě je nejprve ověřeno, obsahuje-li slovník požadovaný klíč (*car*, ř. 2) a pouze pokud ano, jsou klíč i hodnota vypsány do konzolového okna (ř. 3). V případě, že bychom se pokusili tímto způsobem přečíst hodnotu pro klíč, který se ve slovníku nenachází (museli bychom vynechat kontrolu na ř. 2), program by vyvolal výjimku, oznamující, že „daný klíč není ve slovníku k dispozici“.

Následující kód ukazuje, jak v témže formátu vypsát celý obsah (všechny položky) slovníku, dvěma různými způsoby.



```
1: // Vypsání celého slovníku (procházení jeho párových položek)
2: foreach (var slovo in slovník)
3:     Console.WriteLine("{0} = {1}", slovo.Key, slovo.Value);
4:
5: // Vypsání celého slovníku (procházení jeho klíčů)
6: foreach (var key in slovník.Keys)
7:     Console.WriteLine("{0} = {1}", key, slovník[key]);
```

První přístup (ř. 2–3) postupně prochází jednotlivé položky slovníku<sup>9</sup> a z nich u každé vypíše jejich klíč a hodnotu. Druhá varianta (ř. 6–7) ze slovníku získá seznam všech jeho klíčů (vlastnost slovníku **Keys**) a až ten se postupně zpracuje v cyklu **foreach**. Pro každý klíč pak lze ze slovníku snadno získat hodnotu její indexací (`slovník[key]`).

A na závěr si ukážeme, jak vytvořit slovník z posloupnosti, seznamu, či jiné kolekce hodnot, pomocí převodních metod<sup>10</sup>.

```
1: // Posloupnost s 10ti členy (indexy 0-9) řady hodnot 1-10
2: var a = new int[10];
3: for (int i = 0; i < a.Length; i++)
4:     a[i] = i+1;
5: // Převod posloupnosti na slovník čísel a logických hodnot
6: Dictionary<int, bool> dict = a.ToDictionary(
7:     k => k,           // Klíč = číslo,
8:     v => v % 2 == 0 // Hodnota = je toto číslo sudé?
9: );
10: // Vypsání slovníku "[1, False], [2, True], [3, False], ..."
11: Vypis(dict);
```

---

<sup>9</sup> Položky slovníku jsou hodnoty datového typu (struktury) **KeyValuePair**, která má vlastnosti **Key** a **Value**.

<sup>10</sup> Slovník ze seznamu by se klasicky vytvořil procházením seznamu a postupným přidáváním jeho položek do slovníku.



Nejprve tedy vytvoříme posloupnost **a** (ř. 2) s deseti členy a naplníme ji řadou hodnot (ř. 3–4) od 1 (na indexu 0) do 10 (na indexu 9). Následně prostřednictvím rozšiřující (*extension*) metody pro kolekce všeho druhu (včetně 1D polí) **ToDictionary** (více viz kap. 2.5), převedeme členy této posloupnosti na slovník (ř. 6–9). Jelikož tyto členy jsou obvyčejné různé celočíselné hodnoty, tak jejich hodnoty použijeme jako klíče položek slovníku (ř. 7) a jako slovníkové hodnoty typu **bool** použijeme informaci, je-li klíčové číslo sudé (**true**) či nikoli (**false**; ř. 8).

Vzniklý slovník si poté vypíšeme do řádku pomocí univerzální metody **Vypis** (viz závěr kap. 2.3). Tato metoda dokáže vypsát libovolný seznam hodnot, přičemž slovník je z tohoto pohledu seznamem hodnot typu **KeyValuePair**, jenž jako svou textovou reprezentaci vrací v hranatých závorkách klíč a hodnotu (`[key, value]`, tj. v tomto případě např. `[1, False]`).

## 2.5 LINQ

LINQ není dalším typem seznamu, ale alternativní možností, jak s nimi pracovat. LINQ je zkratkou *Language Integrated Query* (integrováný dotazovací jazyk) a zde si ukážeme konkrétně *LINQ to Objects*, tedy dotazování na objekty v paměti<sup>11</sup>.

Existují dva základní způsoby, jak lze LINQ v kódu používat. První, jednodušší a také ten, který si tu blíže představíme je voláním rozšiřujících (*extension*) metod (obvykle pro báзовou třídu všech kolekcí **Enumerable**) ze jmenného prostoru **System.Linq**, které se volají buď bez parametrů nebo nezářídka s využitím tzv. *lambda výrazů*.

Druhý způsob zápisu LINQ dotazů je pak podobný stylu SQL dotazů a je určen spíše pro řešení složitějších úloh. Následující kód porovnává oba zápisy pro vzestupné seřazení členů posloupnosti **a** dle jejich absolutní hodnoty.

```
1: // Vytvoření a naplnění posloupnosti
2: int[] a = { 1, -4, 3, 55, 4, -3, 1 };
3: // LINQ příkaz pro seřazení členů dle jejich absolutní hodnoty
4: var x1 = from p in a
5:           orderby Math.Abs(p)
6:           select p;
7: // Stejný příkaz zapsaný extension metodou s lambda výrazem
8: var x2 = a.OrderBy(p => Math.Abs(p));
```

Tento kód tedy do proměnných **x1** a **x2** přiřadí seznamy (specifického typu, tzv. anonymní třídy) stejně uspořádaných celočíselných položek o týchž hodnotách, jako byly v původní posloupnosti **a**. Pořadí položek obou

---

<sup>11</sup> Existuje také *LINQ to SQL* (dotazování nad databází) či *LINQ to XML* (dotazy prohledávající XML strukturu) apod., přičemž zápis dotazů v C# je podobný, odlišný je zdroj dat, způsob jejich načítání a procházení.

seznamů přitom bude seřazeno dle jejich absolutní hodnoty (1, 1, 3, -3, -4, 4, 55). Pro zpracování členů byla v obou případech použita proměnná **p**. První seřazení (ř. 4–6) bylo zapsáno prostřednictvím mini-jazyka LINQ a druhé (ř. 8) prostřednictvím rozšiřující metody a lambda výrazu (obsahuje operátor  $\Rightarrow$ ). Algoritmy na pozadí obou příkazů jsou nicméně totožné.

### 2.5.1 Agregční funkce

Vezměme to ale postupně. Jak by vypadaly základní algoritmy nad posloupností (viz kap. 2.1.2 a 2.1.3) s využitím LINQ metod?

```
1: // Agregční funkce
2: Console.WriteLine("součet = {0}", a.Sum());
3: Console.WriteLine("počet = {0}", a.Count());
4: Console.WriteLine("průměr = {0}", a.Average());
5: Console.WriteLine("minimum = {0}", a.Min());
6: Console.WriteLine("maximum = {0}", a.Max());
```

Základní statistiky o libovolném číselném seznamu či posloupnosti tedy zjistíme takto snadno. Pokud potřebujeme do výpočtu zařadit třeba nějakou podmínku, bude již nutné zapojit také lambda výrazy.

```
1: // Počty hodnot s určitou vlastností
2: Console.WriteLine("lichých = {0}", a.Count(x => x % 2 == 1));
3: Console.WriteLine("sudých = {0}", a.Count(x => x % 2 == 0));
```

Počet hodnot s určitou vlastností, v tomto případě lichých (ř. 2) a sudých (ř. 3) hodnot, tak zjistíme také celkem snadno. Základní struktura lambda výrazu je tedy taková, že si zvolíme proměnnou, do níž se postupně promítanou všechny položky seznamu a za operátor  $\Rightarrow$  zapíšeme výraz, který vrátí hodnotu požadovaného datového typu. V tomto případě to byl **bool**, tedy má-li se daná položka seznamu do počtu započítat (**true**) či nikoli (**false**).

V následujícím příkladu je pro funkci **Sum** jako výsledek lambda výrazu požadována číselná hodnota, tj. číslo, které se má k součtu přičíst.

```
1: // Součet drhých mocnin
2: Console.WriteLine("suma čtverců = {0}", a.Sum(x => x * x));
```

Můžeme také najednou otestovat celý seznam/posloupnost, jestli všechny prvky (**All**, ř. 2) splňují určitou podmínku, popř. existuje-li alespoň jeden (**Any**, ř. 3), který ji splňuje. Zajímá-li nás obsahuje-li seznam nějakou konkrétní hodnotu, lze použít metodu **Contains**. Výstupem všech tří metod je logická hodnota, tedy **true** nebo **false**.

```
1: // Test celého seznamu
2: Console.WriteLine("vše je sudé: {0}", a.All(x => x % 2 == 0));
3: Console.WriteLine("něco lichého:{0}", a.Any(x => x % 2 == 1));
4: Console.WriteLine("je tam 5: {0}", a.Contains(5)? "ANO": "NE");
```

### 2.5.2 Pod části seznamu

Kromě agregačních funkcí, které jako svůj výsledek vracejí jedinou hodnotu, jsou zde i metody, jejichž výstupem je opět seznam. Vstupní seznam přitom není nijak dotčen a vrácen je vždy nový (změněný) seznam.

#### Filtrování

Metoda **Where** filtruje prvky seznamu a ve výstupu ponechává pouze ty, které projdou zadanou podmínkou (zůstanou jen sudé hodnoty). (Metodu **Vypis** jsme vytvořili dříve, v závěru kap. 2.3.)

```
1: // Vypsat sudé prvky
2: Console.Write("sudé hodnoty: ");
3: Vypis(a.Where(x => x % 2 == 0));
```

Návratovou hodnotou metody **Where** je tedy vyfiltrovaný seznam položek, v tomto případě s pouze sudými hodnotami. Díky tomu lze rovnou pokračovat dalšími metodami a výsledný seznam nadále upravovat. Následující kód např. ukazuje, jak z takto vyfiltrovaných sudých hodnot spočítat (a rovnou i vypsát) jejich průměr.

```
1: // Výpočet průměru z pouze těch prvků, které jsou dvojciferné
2: Console.WriteLine("průměr dvojciferných = {0}",
    a.Where(x => x >= 10 && x < 100).Average());
```

Kromě filtrování na základě podmínky je zde ještě funkce **Distinct**, která ze seznamu vyřadí veškeré položky s duplicitními hodnotami.

```
1: // Výpis jedinečných hodnot (každá v seznamu pouze 1x)
2: Vypis(a.Distinct()); // 1, 1, 2, 3, 3 => 1, 2, 3
```

## Řazení

Funkci řazení již jsme nastínili v úvodní ukázce. Jejím povinným parametrem je tedy *lamda* výraz, který stejně jako v předchozích případech načítá do proměnné (**x**) jednotlivé položky, avšak výsledkem tohoto nemá být logická hodnota jako u **Where** (**true** pro zařazení do výsledného seznamu, **false** pro vyřazení), ale hodnota, podle které se má posloupnost řadit. V případě posloupnosti čísel použijeme tedy číslo samotné. Následující kód ukazuje, jak seznam seřadit vzestupně (**OrderBy** – od nejnižšího čísla po nejvyšší) a sestupně (**OrderByDescending** – od nejvyššího čísla po nejnižší).

```
1: // Vypsání seřazených hodnot
2: Console.Write("vzestupně seřazená čísla: ");
3: Vypis(a.OrderBy(x => x));
4: Console.Write("sestupně seřazená čísla: ");
5: Vypis(a.OrderByDescending(x => x));
```

Pokud by položky seznamu měly více hodnot, podle kterých by je bylo možné řadit (např. seznam lidí řazený dle příjmení a křestního jména), pak lze na výsledný seřazený seznam navázat druhým stupněm řazení pomocí metody **ThenBy**. Následující kód ukazuje seřazení prvků dle jejich absolutní hodnoty a v případě shody této absolutní hodnoty (např.  $|-4| == |4|$ ) by se nejprve zařadily hodnoty kladné a až poté jejich záporné varianty (řazeno dle  $-x$ , tzn. např.  $-4 < -(-4)$ ).

```
1: // Vícestupňové řazení
2: Vypis(a.OrderBy(x => Math.Abs(x)).ThenBy(x => -x));
```

Pro úplnost doplňme, že kromě **ThenBy** existuje také **ThenByDescending**, přičemž za výsledek tohoto druhého stupně řazení lze i dále přidat libovolný počet dalších stupňů řazení pomocí těchto dvou metod.

### Pod částí dle pozice

Část seznamu lze extrahovat nejen za základě nějakého pravidla pro hodnoty jednotlivých prvků, ale třeba i na základě jejich pozice. Metoda **Take** vrací pod část seznamu s prvky převzatými z výchozího seznamu (**a**) z jeho počátku (od prvního, tj. index 0) v počtu definovaném ve vstupním parametru této metody.

```
1: // Pod část ze začátku seznamu
2: Console.WriteLine("první tři prvky: ");
3: Vypis(a.Take(3));
4: // Pod část ze začátku nejprve seřazeného seznamu
5: Console.WriteLine("tři nejmenší prvky: ");
6: Vypis(a.OrderBy(x => x).Take(3));
7: // Seznam s jedním (nejmenším) prvkem
8: Console.WriteLine("první nejmenší (minimum): ");
9: Vypis(a.OrderBy(x => x).Take(1)); // Seznam s prvkem a.Min()
```

V ukázce byla metoda **Take** použita nejen samostatně (ř. 3), ale také v kombinaci s řazením (**OrderBy**, ř. 6 a 9). Díky tomu je seznam nejprve seřazen a až z jeho seřazené verze vráceny první (nejnižší) tři (ř. 6), popř. první (minimum, ř. 9) prvek.

Pokud bychom ovšem potřebovali prvky z konce seznamu, pak je tu metoda **Skip**. Ta přeskočí zadaný počet prvků a vrátí pod část seznamu od následujícího prvku za tímto počtem, až do jeho konce. Např. tedy z 1, 2, 3, 4, 5 vrátí `Skip(2)` seznam 3, 4, 5.

```
1: // Pod část seznamu od určitého prvku do konce seznamu
2: Console.WriteLine("bez prvních tří prvků: ");
3: Vypis(a.Skip(3));
4: // Pod část seznamu z jeho středu
5: Console.WriteLine("čtvrtý a pátý: ");
6: Vypis(a.Skip(3).Take(2)); // Od indexu 3 násl. 2 prvky
```

V ukázce je opět kromě základního použití metody **Skip** (ř. 3) použita také varianta kombinující dvě metody do sebe (ř. 6). V tomto případě tedy přeskočíme první tři prvky (indexy 0, 1, 2) a ze vzniklého zbytku seznamu vezmeme pouze první dva prvky (původní indexy 3, 4).

Obě metody pak mají ještě variantu s **-While** (**TakeWhile** a **SkipWhile**), které fungují podobně (vrací pod část seznamu z jeho začátku/konce), avšak prvek, na kterém mají skončit/začít neurčujeme konkrétním počtem požadovaných/přeskočených položek, ale podmínkou pro první takový.



```
1: // Pod část od začátku po prvek s určitou vlastností (bez něj)
2: Console.WriteLine("hodnoty před prvním výskytem sudého čísla: ");
3: Vypis(a.TakeWhile(x => x % 2 == 1));
4: // Pod část od prvku s určitou vlastností (včetně) do konce
5: Console.WriteLine("hodnoty od prvního sudého čísla: ");
6: Vypis(a.SkipWhile(x => x % 2 == 1));
```

Podmínka pro určení dělicího prvku musí být koncipována podobně jako u cyklu **while** (viz kap. 1.9.1), tedy dokud platí tak u **TakeWhile** (ř. 3) prvky bereme, u **SkipWhile** (ř. 6) dokud platí, prvky přeskakujeme.

### 2.5.3 Jeden prvek ze seznamu

Výsledkem předchozích metod (viz kap. 2.5.2) byl tedy seznam, byť v některých případech s pouze jedním prvkem. Následující metody budou vrátet již přímo tento prvek (jeho hodnotu), takže v případě přiřazení výsledku do proměnné **b** (`var b = ...`) by se k hodnotě nepřistupovalo `b[0]`, ale pouze přes **b**, podobně jako tomu bylo u agregačních funkcí (viz kap. 2.5.1).

Pokud potřebujeme hodnotu prvku z okraje seznamu, tedy první nebo poslední, lze použít bezparametrickou verzi metod **First** (pro první prvek, ř. 3) či **Last** (pro poslední prvek, ř. 4).

```
1: // První/poslední prvek seznamu (u prázdného vyvolá vyjímku)
2: Console.WriteLine("první prvek: {0}", a[0]); // Přes index
3: Console.WriteLine("první prvek: {0}", a.First());
4: Console.WriteLine("poslední prvek: {0}", a.Last());
```

Tyto metody mají ovšem i variantu s parametrem pro lambda výraz, ve kterém lze definovat podmínku, jež bude testována s každým prvkem seznamu, buď od začátku (**First**, ř. 2) nebo od konce (**Last**, ř. 3) a vrácen bude první z prvků, který ji splní.





```
1: // První/posl. prvek seznamu dané vlastnosti (není-li => error)
2: Console.WriteLine("první záporný: {0}", a.First(x => x < 0));
3: Console.WriteLine("poslední záporný: {0}", a.Last(x => x < 0));
```

Tyto metody však mají tu vlastnost, že napevno počítají s tím, že tam takový prvek (první/poslední popř. nějaký s danou vlastností) bude. Pokud tam však není (seznam je prázdný, nebo neobsahuje žádný prvek s danou vlastností), metoda vyvolá výjimku, tedy zaseknutí programu. Pokud s tím program počítá (např. na vyšší úrovni zachytává tyto výjimky pomocí **try-catch** bloku, viz kap. 1.3, a adekvátně na ně reaguje), tak je vše v pořádku.

Má-li ovšem program pokračovat i v případě, že požadovaný prvek nebude nalezen, a místo něj použít výchozí hodnotu pro daný datový typ<sup>12</sup>, pak je tu pro obě metody v obou variantách alternativa s postfixem **–OrDefault**.

```
1: // První/poslední prvek seznamu (u prázdného vrátí 0)
2: Console.WriteLine("první prvek: {0}", a.FirstOrDefault());
3: Console.WriteLine("poslední prvek: {0}", a.LastOrDefault());
4: // První/poslední záporný prvek seznamu (neexistuje-li vrátí 0)
5: Console.WriteLine("první záporný: {0}",
    a.FirstOrDefault(x => x < 0));
6: Console.WriteLine("poslední záporný: {0}",
    a.LastOrDefault(x => x < 0));
```

V případě, že bychom potřebovali prvek na určitém indexu, lze krom jeho indexace (`b[0]`) použít i metodu **ElementAt**, čili *prvek na* indexu zadaném jako vstupní parametr, a především pak její variantu, která potlačí případnou výjimku při dotazu na index mimo hranice seznamu a vrátí výchozí (**default**) hodnotu pro daný datový typ.

---

<sup>12</sup> Výchozí hodnotou pro čísla je 0, pro **string** prázdný textový řetězec (""), pro **bool** je to **false**, u referenčních typů **null** atd.

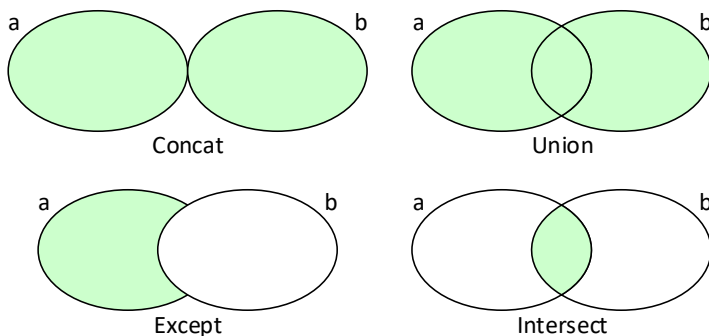


```
1: // Prvek seznamu na daném indexu (neexistuje-li => error)
2: Console.WriteLine("třetí prvek: {0}", a[2]); // Přes index
3: Console.WriteLine("třetí prvek: {0}", a.ElementAt(2));
4: // Prvek seznamu na daném indexu (neexistuje-li vrátí 0)
5: Console.WriteLine("třetí prvek: {0}", a.ElementAtOrDefault(2));
```

## 2.5.4 Spojování seznamů

Máme-li seznamy dva (např. **a** a **b**) a jsou-li téhož datového typu, lze jejich prvky sloučit do jednoho nového seznamu. Pro příkaz `a.METODA(b)` máme k dispozici hned několik metod, z nichž ovšem každá prvky obou seznamů kombinuje odlišným způsobem:

- **Concat** – spojí oba seznamy za sebe a žádných duplicit si nevšímá.
- **Union** – spojí seznamy tak, aby tam každá položka téže hodnoty byla pouze jednou, což se týká nejen stejných hodnot v obou množinách, ale i duplicit v každé z nich (ekvivalent `a.Concat(b).Distinct()`).
- **Except** – odebere z prvního seznamu (**a**) veškeré prvky, které jsou zároveň obsaženy ve druhém seznamu (**b**) a každý ponechá pouze jednou (ekvivalent `a.Where(x => !b.Contains(x)).Distinct()`).
- **Intersect** – společný průnik seznamů, ponechá jen ty položky, které se vyskytují v obou spojovaných seznamech, avšak každou pouze jedenkrát (ekvivalent `a.Where(x => b.Contains(x)).Distinct()`).



Následující ukázka na dva seznamy (posloupnosti) aplikuje všechny čtyři výše popsané metody spojení, přičemž v komentářích jsou uvedeny jejich výsledky.

```
1: // Spojování seznamů
2: var a = new[] { 1, 2, 2, 3, 4 };
3: var b = new[] { 2, 3, 5, 5 };
4: Console.WriteLine("Concat: ");
5: Vypis(a.Concat(b));           // 1, 2, 2, 3, 4, 2, 3, 5, 5
6: Console.WriteLine("Union: ");
7: Vypis(a.Union(b));           // 1, 2, 3, 4, 5
8: Console.WriteLine("Except: ");
9: Vypis(a.Except(b));          // 1, 4
10: Console.WriteLine("Intersect: ");
11: Vypis(a.Intersect(b));       // 2, 3
```

### 2.5.5 Praktické příklady s LINQ

Na závěr uveďme ještě několik praktických příkladů, které lze velmi efektivně řešit právě díky LINQ.

Z textového souboru *zdroj.txt* načtěte celočíselné hodnoty, přičemž každá z je na samostatném řádku, ponechte z nich pouze kladné hodnoty, jež vzestupně seřadíte a výsledný seznam uložte ve stejném formátu do nového souboru *vystup.txt*.

```
1: File.WriteAllLines("vystup.txt", // Uložit následující pole:
2:     File.ReadAllLines("zdroj.txt") // Načtení řádků souboru
3:     .Select(x => Convert.ToInt32(x)) // Převod string na int
4:     .Where(x => x > 0)                // Nechat jen kladné hodnoty
5:     .OrderBy(x => x)                  // Seřadit čísla vzestupně
6:     .Select(x => x.ToString())        // Převést int na string
7:     .ToArray());                     // Převést seznam na pole
```



Načtěte od uživatele **slovo** a na jeho základě ze seznamu slov **slovicka** vyberte do nového abecedně seřazeného seznamu **rymy** ta slova, která se rýmují se zadaným slovem, tzn. shodují se v posledních třech písmenech.

```
1: string slovo = Console.ReadLine(); // Načíst slovo
2: string konec = slovo.Substring(slovo.Length-3); // Část s rýmem
3: var rymy = slovicka // Ze seznamu slovicka...
4:     .Where(x => x.EndsWith(konc)) // Pouze slova končící tímto
5:     .OrderBy(x => x) // Abecedně seřadit vzestupně
6:     .ToList(); // Převést na klasický seznam
```

Ze dvou slovníků (**Dictionary**) pro překlad slovíček z angličtiny do češtiny (**EnToCz**) a z němčiny do angličtiny (**DeToEn**) vytvořte třetí slovník pro překlad z češtiny do němčiny (**CzToDe**).

```
1: // Pokusná data dvou vstupních slovníků
2: var EnToCz = new Dictionary<string, string>() {
3:     { "air", "vzduch" },
4:     { "black", "černá" },
5:     { "sea", "moře" }, // ...
6: };
7: var DeToEn = new Dictionary<string, string>() {
8:     { "luft", "air" },
9:     { "schwarz", "black" },
10:    { "meer", "sea" }, // ...
11: };
12: // Vytvoření nového slovníku CZ=>DE
13: var CzToDe = EnToCz.Keys.ToDictionary(
14:     k => EnToCz[k],
15:     v => DeToEn.FirstOrDefault(x => x.Value == v).Key);
16: // Kontrolní výpis nového slovníku
17: Vypis(CzToDe.Select(x => $"{x.Key} = {x.Value}"));
18: // vzduch = luft, černá = schwarz, moře = meer
```



Do seznamu názvů souborů **soubory** s obrázky typu PNG, potřebujeme přidat další název souboru, náhodně vybraný ze složky s obrázky **slozka**, ale tak, aby v tomto seznamu zatím nebyl.

```
1: /// <summary>Přidá do seznamu název náhodného souboru ze složky,  
    který zatím v tomto seznamu není</summary>  
2: public static void Pridat(List<string> soubory, string slozka)  
3: {  
4:     soubory.Add(Directory                // Přidat do seznamu...  
5:         .GetFiles(slozka, "*.png")      // Všechna PNG ve složce  
6:         // Jen názvy část s názvy souborů "C:\obr1.png" => "obr"  
7:         .Select(x => Path.GetFileNameWithoutExtension(x))  
8:         .Where(x => !soubory.Contains(x)) // Není v seznamu  
9:         .OrderBy(x => random.Next())     // Náhodně seřadit  
10:        .FirstOrDefault());              // První z nich  
11: }
```

## 3 Rekurzivní zpracování vícerozměrného prostoru

Fungování rekurzivního zpracování prostoru si ukážeme na praktickém příkladu, a sice generátoru náhodné mapy 2D bludiště a jeho procházení.

### 3.1 Generátor bludiště s „přepážkami“

Bludištěm s přepážkami je myšleno dvourozměrné bludiště (matice čili dvourozměrné pole) s jednotlivými políčky (buňkami), po kterých se dá „procházet“, přičemž mezi sousedícími políčky v jednotlivých čtyřech směrech buď je anebo není **přepážka**, tedy daný směr buď není (přepážka tam je) či je (přepážka tam není) průchozí.

Nejprve nadefinujeme výčet směrů, kterými se dá mezi jednotlivými políčky bludiště pohybovat. Každému směru také přiřadíme unikátní číslo, které bude v jeho binární podobě možné kombinovat s ostatními čísly (směry) tak, abychom z něho vždy byli schopni dekodovat, které všechny směry toto číslo zahrnuje. Použita tedy budou čísla z řady mocniny dvou. Díky tomu bude možné jednotlivé hodnoty výčtu používat nejen samostatně, ale také zkombinovat do proměnné typu **Smer**, tj. např. proměnná typu **Smer** bude mít hodnotu 10, což znamená, že bude zároveň obsahovat hodnoty D a L (2 + 8, binárně 1010), což půjde pro jednotlivé směry snadno ověřovat pomocí metody **HasFlag**, kterou takové výčty mají.

```
1: public enum Smer    // Vlastní typ hodnot (enum) a jejich výčet...
2: {
3:     N = 1,    // 0001 Nahoru
4:     D = 2,    // 0010 Dolů
5:     P = 4,    // 0100 Pravo
6:     L = 8,    // 1000 Levo
7: }
```

Nyní vytvoříme třídu **Bludiste**, která bude zastřešovat všechny potřebné metody. Třída může být statická, jelikož nebude uchovávat žádné stavy ani hodnoty, veškeré operace se vždy vyřeší v rámci jednotlivých metod.

```
1: public static class Bludiste { ... }
```

Ve třídě si nadefinujeme a rovnou i vytvoříme generátor náhodných čísel **Rnd** a pomocné seznamy směrů. Jednak směry (**Smery**) seřazené odshora ve směru hodinových ručiček (12, 3, 6, 9 hodin), a pak druhý seznam **Smery-Nahodne**, který při každém čtení vrátí seznam předchozí, ovšem směry v něm budou při každém čtení tohoto seznamu v jiném náhodném pořadí. Toho je docíleno pomocí LINQ seřazení **OrderBy** dle náhodného čísla vygenerovaného přímo v daném lambda výrazu.

```
1: // Generátor náhodných čísel
2: static Random Rnd = new Random();
3:
4: // Seznam směrů v pořadí směru hodinových ručiček
5: static Smer[] Smery => new[] { Smer.N, Smer.P, Smer.D, Smer.L };
6:
7: // Náhodně (pokaždé jinak) seřazený seznam všech směrů
8: static IEnumerable<Smer> SmeryNahodne
9:     => Smery.OrderBy(s => Rnd.NextDouble());
```

Dále si ještě připravíme pomocný slovník **Protismer**, který bude překládat každý ze čtyř směrů (klíč) na jemu opačný (hodnota, např. Nahoru => Dolů, Dolů => Nahoru, ...).



```
1: // Překlad směru na směr opačný
2: static Dictionary<Smer, Smer> Protismer =
3:     new Dictionary<Smer, Smer> {
4:         { Smer.N, Smer.D },
5:         { Smer.D, Smer.N },
6:         { Smer.P, Smer.L },
7:         { Smer.L, Smer.P },
8:     };
```

Metoda **Vytvor** si nejprve vytvoří proměnnou **mapa** typu dvourozměrné pole celých čísel o rozměrech určených vstupními parametry (**radku** a **sloupcu**), do které vygeneruje náhodné bludiště zavoláním metody **ZpracujPolicko**, kterou vytvoříme později, s výchozími souřadnicemi 0,0 (levý horní roh).

```
1: /// <summary>Vygeneruje bludiště zadaných rozměrů</summary>
2: public static int[,] Vytvor(int radku, int sloupcu)
3: {
4:     var mapa = new int[radku, sloupcu]; // Vytvoření 2D pole
5:     ZpracujPolicko(0, 0, mapa); // Vygenerování mapy z 0,0
6:     return mapa; // Vrácení vygenerované mapy
7: }
```

Mapa (matice **mapa**) bude na začátku obsahovat samé nuly (výchozí hodnota pro **Int32**), což bude hodnota označující dosud nezpracovaná pole se všemi čtyřmi přepážkami do všech možných směrů (nahoru, dolů, doleva a doprava) uzavřenými. Po zpracování bude každý z členů matice obsahovat nenulové číslo, které v sobě bude kódovat, přepážky kterých všech směrů jsou v něm otevřeny, tj. do kterých směrů lze přejít na sousední pole.

Ještě před vytvořením metody pro zpracování políčka si připravíme dvě pomocné metody, které v ní pak využijeme. První z nich bude metoda **Posun**,



kteřá na vstupu dostane aktuální pozici (**posY** a **posX**) na mapě a směr (**směr**), ve kterém se chceme posunout. Ve výstupních proměnných (**newY** a **newX**) pak vrátí souřadnice sousedního políčka, na které byl posun směrován, a zároveň také jako výstupní hodnotu metody vrátí informaci o tom, jestli je tato nová souřadnice platnou (nachází se v mezích mapy), či nikoli, a nelze ji tudíž použít (pokus o čtení hodnoty mapy s indexy mimo její rozsah by vyvolal výjimku).

```
1: /// <summary>Posune se z pozice pos ve směru směr na pozici new
2: /// a vrátí bool, jestli nová pozice není mimo mapu</summary>
3: static bool Posun(int[, ] mapa, int posY, int posX, Smer smer,
4:                   out int newY, out int newX)
5: {
6:     // Výchozí hodnoty (pozice)
7:     newY = posY;
8:     newX = posX;
9:     // Posun v daném směru
10:    switch (smer)
11:    {
12:        case Smer.N: newY--; break;
13:        case Smer.D: newY++; break;
14:        case Smer.L: newX--; break;
15:        case Smer.P: newX++; break;
16:        default: return false; // Nemožné
17:    }
18:    // Je nová souřadnice v rozmezí rozměrů mapy?
19:    return newY >= 0 && newY < mapa.GetLength(0) &&
20:           newX >= 0 && newX < mapa.GetLength(1);
21: }
```

Druhá pomocná metoda **OtevriPrepazku** bude mít za úkol v políčku se souřadnicemi **posY** a **posX** zrušit přepážku tímto směrem a u souseda se souřadnicemi **newY** a **newX** zrušit přepážku vedoucí na toto políčko (opačný

směr než před tím). Toto zrušení přepážek je realizováno binární funkcí **OR** (značeno jednou svislou čarou "|"), která zkombinuje dvě celá čísla (**int**) do jednoho, a to na binární úrovni (např.  $0010 \mid 0100 = 0110$ , tj.  $2 \mid 4 = 6$ ).

```
1: /// <summary>Nastaví políčku výchozímu (pos) i sousednímu
2: /// cílovému (new), že přepážka mezi nimi není</summary>
3: static void OdeberPrepazku(int[, ] mapa, int posY, int posX,
                           int newY, int newX, Smer smer)
4: {
5:     mapa[posY, posX] |= (int)smer;
6:     mapa[newY, newX] |= (int)Protismer[smer];
7: }
```

Metoda **ZpracujPolicko** zpracovává jedno políčko (buňku) referenčně předávané mapy **mapa** na zadaných souřadnicích **posY** a **posX** a toto zpracování rekurzivně posílá dál všem jeho dosud nezpracovaným sousedům v náhodném pořadí.

Náhodného zpracování jednotlivých sousedů a přechodů na ně je docíleno procházením dříve nadefinovaného pomocného seznamu **SmeryNahodne**, který při každém čtení vrací všechny čtyři směry v jiném náhodném pořadí. Díky zamíchání seznamu pak mohou být směry zpracovány již v klasickém **foreach** cyklu.

V něm jsou nejprve pomocí dříve definované metody **Posun** určeny souřadnice sousední buňky, na kterou daný směr vede (**newY**, **newX**). Pokud tyto souřadnice nejsou mimo rozsah mapy a toto sousední políčko dosud nebylo zpracováno (jeho hodnota je 0), je otevřen průchod mezi oběma políčky pomocí metody **OdeberPrepazku**.

Po úspěšném vyřešení daného směru je zpracování rekurzivně předáno na políčko, jež v tomto směru sousedí, a až pak přijde na řadu řešení směru

následujícího. Díky tomu jsou postupně zpracována všechna políčka na mapě (v matici), tzn. každé z nich je propojeno náhodným průchodem (průchody) k „cestě“ vedoucí po celé mapě.

```
1: /// <summary>Zpracuje políčko bludiště a rekurzí jej pošle dál
2: /// svým nenavštíveným sousedům v náhodném pořadí</summary>
3: static void ZpracujPolicko(int posY, int posX, int[,] mapa)
4: {
5:     // Postupné zpracování čtyř sousedních políček v náh. pořadí
6:     foreach (var smer in SmeryNahodne)
7:     {
8:         // Určení souřadnic cílového políčka v daném směru
9:         if (!Posun(mapa, posY, posX, smer,
10:                    out int newY, out int newX))
11:             continue; // Souřadnice jsou mimo mapu = tedy ne
12:             // Test, nebylo-li již toto pole zpracováno
13:             if (mapa[newY, newX] != 0)
14:                 continue; // Tam už byl
15:             // Zrušení přepážky mezi tímto a cílovým políčkem
16:             OdeberPrepazku(mapa, posY, posX, newY, newX, smer);
17:             // Předání zpracování tomuto sousednímu políčku
18:             ZpracujPolicko(newY, newX, mapa);
19:     }
```

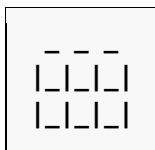
Výstupem takového zpracování je plnohodnotné bludiště, v němž je přístupné každé z jeho políček (žádné není uzavřeno ze všech čtyř stran, ani žádná oblast není zcela uzavřena) a zároveň zde není žádné pole zcela otevřené (není zde zbytečný volný prostor ani 2x2). V bludišti se také nevyskytují smyčky, tj. žádné přepážky, které by nebyly propojené s okolní hranicí mapy, či jinak řečeno, není zde žádný útvar přepážek, kolem kterých by se dalo donekonečna kroužit. Tudíž cesta bez vracení z libovolného políčka do kteréhokoli jiného vede vždy právě jedna.

## 3.2 Vykreslování přepážkové mapy bludiště do konzole

Mapa bludiště je tedy vygenerována, ovšem jakožto matici čísel si ji moc dobře neprohlédneme. Vytvoříme si tedy další metodu **Vypis**, která nám na základě této matice dokáže mapu přehlednou pro člověka vykreslit do konzole, pomocí standardních znaků podtržítka (|) a svislé čáry (|).

Při postupném vypisování mapy bude stačit vykreslit horní a levý okraj celé mapy, a uvnitř pro každou buňku již pouze jejich dolní a pravé přepážky. Díky tomu následující políčka budou mít levé a horní přepážky již takto vyřešeny od jejich dříve vykreslených sousedů.

Podtržítko je prázdným znakem s čarou až u své spodní hrany a lze jej tak použít pro vykreslení jak horizontálně průchozího prostoru, tak i spodní přepážky zároveň. Znak svislé čáry má tuto čáru horizontálně uprostřed tohoto znaku a s podtržítkem do jednoho znaku vypsát nelze, čímž nevyhovuje naznačení vertikální průchodnosti pole s přepážkou napravo, a musí se tak ve výpisu v konzoli zkombinovat se znakem mezery (tj. " |", "| |", "| " nebo " ").



Vzhledem k tomu způsobu výpisu bude mít mapa bludiště ve výpisu do šířky dvojnásobný počet znaků, než je jeho skutečná šířka, zatímco do výšky bude počet znaků korespondovat s jeho skutečnou výškou. To je nicméně v důsledku i výhodnější, neboť znaky v konzoli mají cca dvojnásobnou výšku než šířku. Pro každý sloupec tedy bude vypsáno nejprve buď podtržítko (dolů projít nelze), nebo mezera (dolů projít lze) a následně (druhý znak) buď svislá čára (doprava projít nelze), či mezera (doprava projít lze).



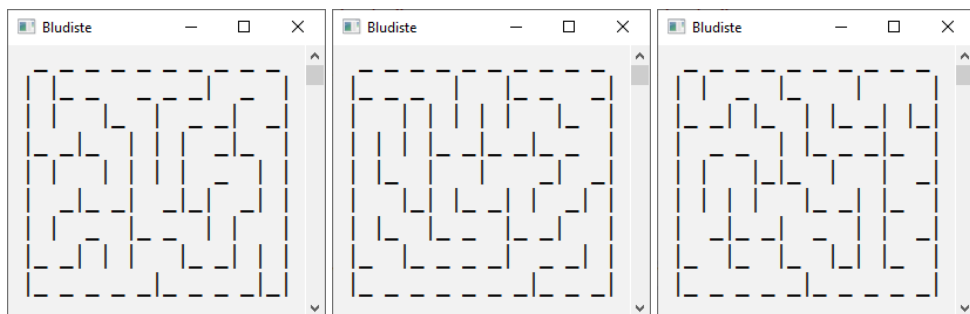
```
1: /// <summary>Vypsání přepážkového bludiště do konzole</summary>
2: public static void Vypis(int[,] mapa)
3: {
4:     // Horní hranice mapy
5:     Console.Write(" ");
6:     for (int j = 0; j < mapa.GetLength(1); j++)
7:         Console.Write(" _");
8:     Console.WriteLine();
9:
10:    // Řádky bludiště
11:    for (int i = 0; i < mapa.GetLength(0); i++)
12:    {
13:        Console.Write(" |");    // Levá hranice mapy
14:        // Sloupce bludiště
15:        for (int j = 0; j < mapa.GetLength(1); j++)
16:        {
17:            Smer s = (Smer)mapa[i, j];
18:            Console.Write(s.HasFlag(Smer.D) ? " " : "_");
19:            Console.Write(s.HasFlag(Smer.P) ? " " : "|");
20:        }
21:        Console.WriteLine();
22:    }
23: }
```

### 3.3 Hlavní řídicí program

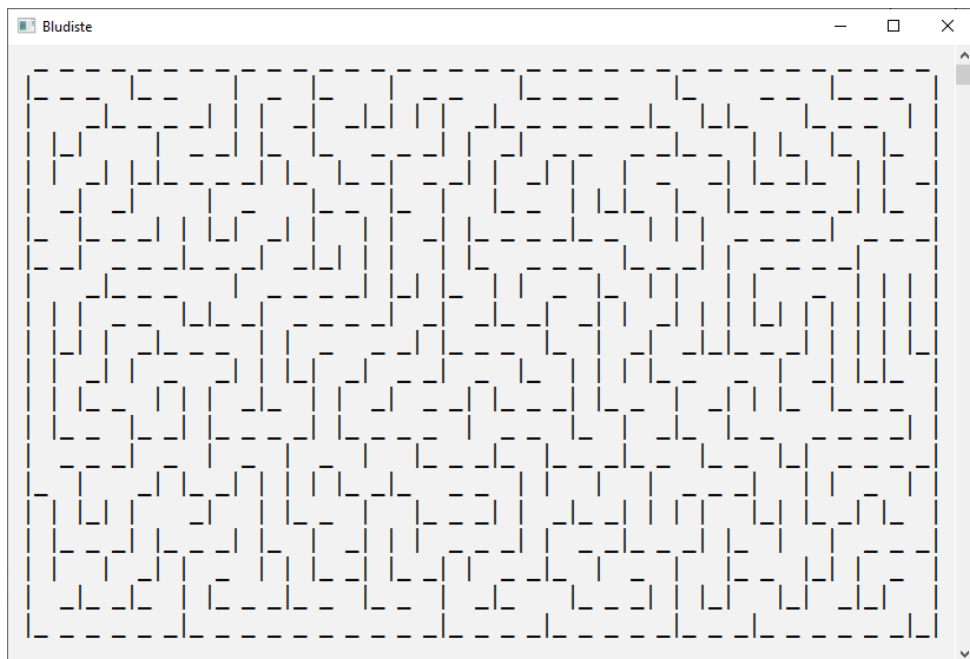
Statická třída **Bludiste** nyní již obsahuje vše potřebné, můžeme tedy vyzkoušet bludiště vygenerovat a vypsát. V hlavní metodě aplikace (**Program.Main**) zavoláme nejprve metodu **Vytvor**, která nám vrátí mapu bludiště jako 2D pole (matici) celých čísel (**int[,]**) zadaných rozměrů a tuto mapu necháme následně vypsát do konzole metodou **Vypis**. Na konec metody přidáme příkaz **Console.ReadLine**, který zastaví běh programu (jeho okamžité vypnutí), dokud nestiskneme klávesu Enter.

```
1: class Program
2: {
3:     static void Main(string[] args)
4:     {
5:         int[,] mapa = Bludiste.Vytvor(8, 10); //Vygenerování mapy
6:         Bludiste.Vypis(mapa);                // Vypsání mapy s přepážkami
7:         Console.ReadLine();                  // Vyčkání na stisk Enteru
8:     }
9: }
```

Výstup v konzolovém okně (barvy byly přepnuty na světlé pozadí a tmavý text) pak může vypadat například takto (3 různá spuštění pro rozměr 8x10).



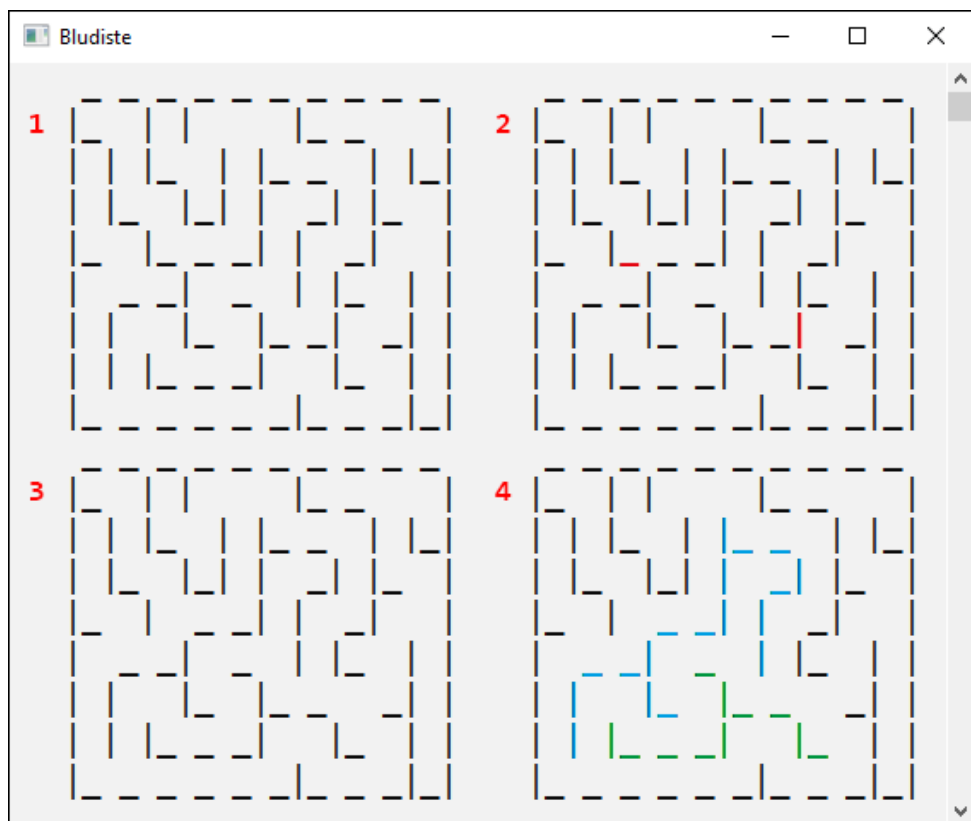
Větší bludiště (20x35) pak může vypadat například následovně.



### 3.4 Přidání smyček

Jak již bylo uvedeno výše, vygenerované bludiště neobsahuje žádné smyčky (zkratky), čili z libovolného místa bludiště do kteréhokoli jiného místa, nevstoupíme-li na žádné místo více než 1x, vede právě jedna cesta.

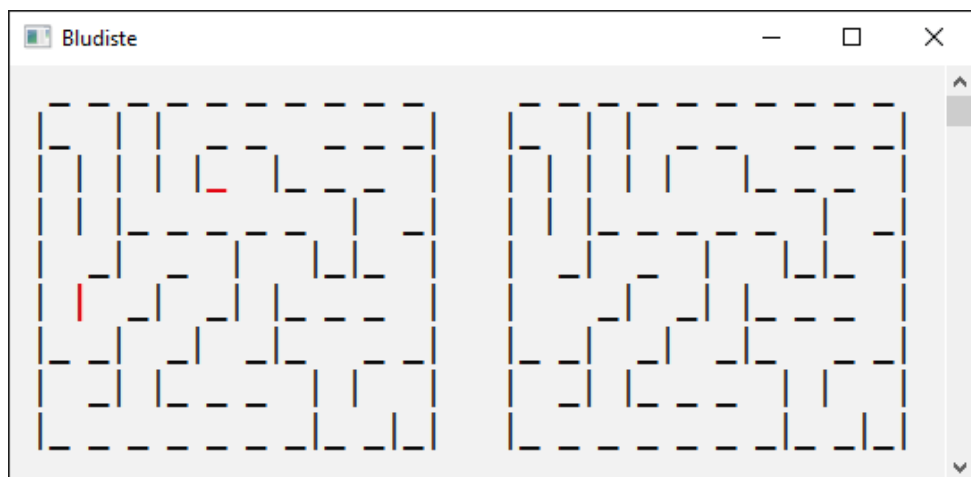
Pokud bychom bludiště chtěli o tyto smyčky obohatit, teoreticky by mohlo stačit náhodně nějaké přepážky odebrat. Tento postup ilustruje následující obrázek ve čtyřech krocích.



V prvním kroku máme tedy klasickou mapu vygenerovaného bludiště bez smyček. Ve druhém kroku náhodně vybereme dvě přepážky pro odebrání (zvýrazněné červeně). Ve třetím kroku tyto přepážky odebereme, čímž vzniknou dvě smyčky. Ty jsou barevně zvýrazněné (modře a zeleně) ve čtvrté mapě.

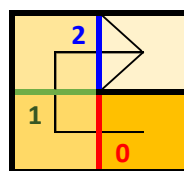
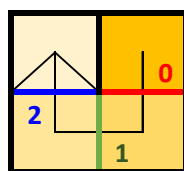
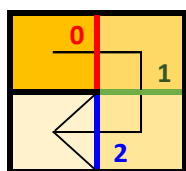
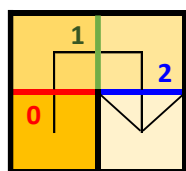
Takto hezky to ovšem funguje pouze v ideálním případě, „když to náhodou vyjde“. Přepážky totiž nelze odebírat zcela náhodně, jelikož by nám zmizení některých z nich mohlo v mapě bludiště vytvořit nehezky prázdné prostory, jak ukazuje následující obrázek.





V tomto případě již výběr přepážek (v mapě vlevo jsou zvýrazněné červeně) pro odebrání tak šťastný nebyl. Po jejich zmizení totiž nejen že žádné smyčky v bludišti nevzniknou, ale mapa naopak ztratí na kráse tím, že bude obsahovat zcela prázdná místa o rozměrech 2x2 buňky (viz mapa vpravo), které vyloženě bijí do očí.

Než tedy odebereme nějakou přepážku z bludiště, bude zapotřebí zkontrolovat, jestli tím nevznikne takovýto prostor, protože to by zároveň indikovalo, že odebrání této přepážky je kontraproduktivní. Kontrolu toho, je-li určitá přepážka v prostoru 2x2 jediná a nelze ji tudíž odebrat, ilustruje následující obrázek pro všechny čtyři varianty (v prostoru 2x2 mohou být maximálně 4 přepážky).



Pokud lze projít z výchozí buňky ve třech krocích ve směru hodinových ručiček až na buňku, která je na druhé straně přepážky, jež se chystáme odebrat, pak tuto přepážku odebrat nemůžeme. Za tímto účelem si připravíme dvě metody, které nám budou přepínat aktuální směr na ten následující. Jedna ve směru hodinových ručiček a druhá v jeho protisměru.

```
1: // Další směr ve směru hodinových ručiček s nekonečným otáčením
2: static Smer DalsiSmer(Smer smer) => smer == Smer.L ? Smer.N
   : Smery[Array.IndexOf(Smery, smer) + 1];
3:
4: // Předchozí směr (další v protisměru) hodinových ručiček
5: static Smer PredchoziSmer(Smer smer) => smer == Smer.N ? Smer.L
   : Smery[Array.IndexOf(Smery, smer) - 1];
```

Obě metody by bylo samozřejmě možné nahradit opět slovníky, které by pro každý směr vracely jeho následující/předchozí variantu, ale vzhledem k tomu, že v poli **Smery** máme seznam směrů již seřazený ve směru hodinových ručiček, můžeme jej využít, pro každý směr vracet směr s indexem o jednu vyšším, a pouze pro ten poslední (**L**) vrátit zase ten první (**N**). A totéž obráceně pro druhou metodu (pro **N** vracíme **L**, jinak směr s indexem -1 od aktuálního).

Celý proces kontroly (obejití přepážky ve směru hodinových ručiček ve třech krocích) pak provádí následující metoda.



```
1: ///
```

Metoda **JePrepazkaOkolo** v cyklu o třech krocích (0-2) vyjde z výchozí pozice (**posY** a **posX**) ve směru (**smer**) a po každém z kroků se pootočí na další směr (doprava). Pokud po této cestě narazí na přepážku (nebo by měl vyjít z mapy ven, což je ale také přepážka) vrátí **true**, v opačném případě, když tam žádná další přepážka není, vrátí **false**. Výsledek **false** tedy znamená, že obcházenou přepážku odebrat nemůžeme, protože je tam jediná.

Pouhá jedna z kontrol znázorněných na předchozím obrázku však není dostačující, protože ověřuje prostor pouze v jednom směru. Přepážky naštěstí máme jen dvojího typu, a to vertikální a horizontální. Pro každý z těchto typů tedy potřebujeme kontroly dvě: pro vertikální přepážku je třeba ji obejít nad ní a pod ní (v předchozím obrázku 1. a 3. varianta) a pro přepážku horizontální pak musíme postupovat vpravo a vlevo (v předchozím obrázku varianta 2 a 4).

Takovéto obcházení lze přitom spojit do jedné delší „procházky“. Po dokončení první cesty, z její koncové pozice, ve směru pootočeném o jeden zpět (aby se pokračovalo rovně), hned pokračovat a celou kontrolu ještě jednou zopakovat. Tento postup pro oba případy ilustruje následující obrázek.



Tu realizujeme pomocí metody **JePrepazkaOkolo**, a pokud z ní získáme **false**, kontrolu můžeme ukončit se záporným výsledkem, jelikož by v tomto směru odebrání přepážky vytvořilo volný prostor  $2 \times 2$ . V opačném případě ještě týmž postupem zkontrolujeme okolí přepážky ve druhém směru, resp. protáhneme „procházku“ okolo přepážky z druhé strany. Výsledek této druhé kontroly pak již můžeme přímo vrátit, jako celkový výsledek této metody.

Takto tedy můžeme otestovat libovolnou přepážku v celém bludišti, jestli ji lze bez negativních následků odebrat či nikoli. Této možnosti využijeme v následující metodě, která bude mít za úkol odebrat z bludiště přepážek hned několik.

Počet přepážek, které bychom rádi z bludiště odebrali, zadáme do vstupního parametru **pocet** této metody, buď konkrétně (hodnota bude rovna nebo větší než 1), nebo relativně (hodnota bude menší než jedna). Relativní verze hodnoty bude v podstatě procentem z celkového počtu buněk, takže např. u mapy  $5 \times 10$  (tj. 50 buněk) a počtu 0,05 (tj. 5 %) se z mapy bludiště náhodně odeberou 3 přepážky ( $50 * 0,05 = 2,5 \doteq 3$ ).

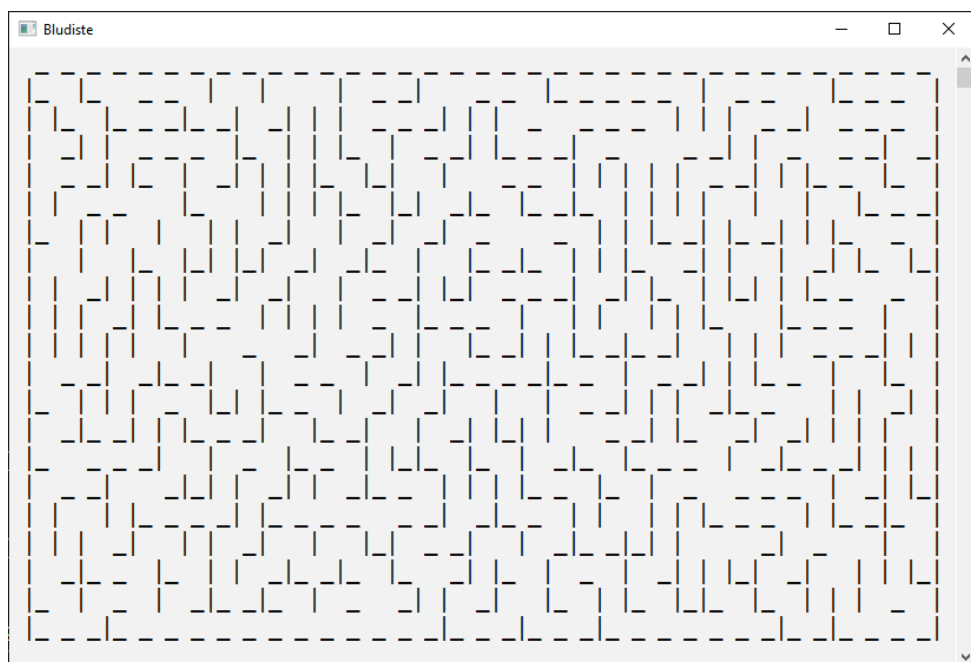


```
1: /// <summary>Přidání smyček (odebrání přepážek) v mapě</summary>
2: public static void PridejSmycky(int[,] mapa, double pocet = .05)
3: {
4:     int zbyvajiciPocet = pocet >= 1 ? (int)pocet :
5:         (int)Math.Round(mapa.Length * pocet); // Počet určen %
6:     while (zbyvajiciPocet > 0)
7:     {
8:         // Vybrat náhodné políčko na mapě
9:         int posY = Rnd.Next(mapa.GetLength(0));
10:        int posX = Rnd.Next(mapa.GetLength(1));
11:        // Zkusit odebrat přepážku v některém z náhodných směrů
12:        foreach (var smer in SmeryNahodne)
13:        {
14:            // Je v tomto směru vůbec ještě přepážka?
15:            if (((Smer)mapa[posY, posX]).HasFlag(smer))
16:                continue; // Není = není co odebírat
17:            // Nevede tento směr vede mimo mapu?
18:            if (!Posun(mapa, posY, posX, smer,
19:                out int newY, out int newX))
20:                continue; // Vede = tedy ne
21:            // Nevznikne smazáním přepážky prázdný prostor 2x2?
22:            if (!LzePrepazkuOdebrat(mapa, posY, posX,
23:                newY, newX, smer))
24:                continue; // Vznikne = tuto přepážku neodebírat
25:            // Odebrat přepážku v daném směru (z obou stran)
26:            OdeberPrepazku(mapa, posY, posX, newY, newX, smer);
27:            zbyvajiciPocet--; // Jedna přepážka byla odebrána
28:            break;          // Z tohoto políčka už neodebírat
29:        }
30:    }
31: }
```

Metoda **PridejSmycky** tedy odebere z mapy bludiště příslušný počet přepážek a tím v něm vytvoří smyčky. Postup odebírání je takový, že v cyklu, který odebrané přepážky počítá, se nejprve vybere náhodná pozice v rámci mapy bludiště. Z ní se v náhodném pořadí zkontrolují postupně všechny čtyři směry, je-li v daném směru vůbec nějaká přepážka, jestli tento směr nevede

mimo mapu a zda odebráním této přepážky nevznikne prázdný prostor 2x2. Pokud všechny tyto kontroly v pořádku projdou, přepážka v tomto směru bude odebrána, to se započítá a celý proces se opakuje na jiné náhodné pozici.

Bludiště o rozměrech 20x35 s přidáním smyček v počtu 5 % (tj. 35) pak může vypadat například následovně.



Takováto úprava mění vlastnosti bludiště hned z několika ohledů. Nyní se lze z určitých míst bludiště na některá jiná místa dostat více možnými cestami, přičemž tyto cesty mohou být různě dlouhé. Pokud budeme bludiště

procházet pomocí pravidla pravé (popř. levé) ruky<sup>13</sup>, na některá místa v bludišti se vůbec nedostaneme, ani když se nakonec vrátíme až do původní pozice. A začneme-li u stěny smyčky, pak budeme pouze „kroužit“ okolo ní. Takže při prohledávání bludiště je zapotřebí postupovat trochu jiným způsobem.

### 3.5 Nalezení nejvzdálenější buňky

Bludiště, ať již s přesmyčkami nebo bez nich, bychom tedy měli. Pokud jej má ale někdo procházet, je zapotřebí stanovit ještě dvě věci: start a cíl. Čili, kudy hráč/luštitel do bludiště bude vstupovat a kam se má dostat, aby bludiště „vyluštil“.

Při generování bludiště jsme začínali v levém horním rohu, tj. na souřadnicích  $[0, 0]$  (v metodě **Vytvor** je volána metoda **ZpracujPolicko** s těmito souřadnicemi). Ovšem stejně jako by generování bludiště bylo možné zahájit ze kterékoli jiné buňky na mapě, můžeme i pro start prohledávání bludiště zvolit libovolnou souřadnici. Jako start lze tedy použít nejen levý horní roh  $[0, 0]$ , ale i kterýkoli jiný roh, střed bludiště anebo i ryze náhodně vygenerované hodnoty pro **Y** a **X** v mezích mapy bludiště.

Umístíme-li ale start na náhodnou pozici, kde bude potom cíl? Můžeme s ním samozřejmě postupovat stejně a umístit jej třeba do protějšího rohu než start, nebo také zcela náhodně. Nicméně mapa bludiště může být taková, že zrovna mezi těmito dvěma buňkami bludiště povede přímá trasa, tudíž nevhodná volba startu a cíle dokáže jinak kvalitní bludiště značně deklasovat.

---

<sup>13</sup> Pravidlo pravé ruky funguje tak, že na začátku přiložíme pravou ruku na stěnu bludiště v místě, kde stojíme a budeme postupovat tak, abychom ruku měli na zdi neustále přiloženou.



Chceme-li být důslední, můžeme pro zvolený start, ať již jej umístíme kamkoli, dohledat cíl, který bude pro danou výchozí buňku v bludišti tou nejvzdálenější. Za tímto účelem vytvoříme metodu, které na vstupu zadáme souřadnice startu (**startY** a **startX**), mapu (**mapa**) a jako výstupní proměnné (**out**) metoda vrátí souřadnice buňky nejvzdálenější od tohoto startu (**konecY** a **konecX**). Návratovou hodnotou metody pak bude vzdálenost mezi startem a cílem, tj. kolik buněk je minimálně třeba projít, než se z jedné této buňky (ze startu) dostaneme do druhé (do cíle).

```
1: public static int NejvzdalenejsiPole(int startY, int startX,
                                     int[,] mapa, out int konecY, out int konecX)
2: {
3:     // Vytvoření matice vzdáleností s výchozími hodnotami -1
4:     var vzdalenosti = new int[mapa.GetLongLength(0),
                               mapa.GetLongLength(1)];
5:     for (int i = 0; i < vzdalenosti.GetLength(0); i++)
6:         for (int j = 0; j < vzdalenosti.GetLength(1); j++)
7:             vzdalenosti[i, j] = -1; // -1 = neřešená buňka
8:     vzdalenosti[startY, startX] = 0; // Jen na startu bude 0
9:     // Určení vzdáleností od startu pro všechny buňky
10:    VzdalenostiOkolo(startY, startX, mapa, vzdalenosti);
11:    // Nalezení souřadnic maxima
12:    (konecY, konecX) = (startY, startX); // Výchozí souřadnice
13:    for (int i = 0; i < vzdalenosti.GetLength(0); i++)
14:        for (int j = 0; j < vzdalenosti.GetLength(1); j++)
15:            if (vzdalenosti[i, j] > vzdalenosti[konecY, konecX])
16:                (konecY, konecX) = (i, j); // Nové maximum
17:    return vzdalenosti[konecY, konecX]; // Vrácení max. vzdál.
18: }
```

Metoda **NejvzdalenejsiPole** nejprve vytvoří matici **vzdalenosti** stejných rozměrů, jako je mapa bludiště (ř. 4) a nastaví všem jejím členům výchozí hodnotu -1 (ř. 5–7). Tato hodnota bude znamením, že buňka bludiště na daných souřadnicích dosud nebyla zpracována. Poté je startovnímu políčku



nastavena vzdálenost 0 (ř. 8) a od něho je zahájeno rekurzivní procházení bludiště pomocí metody **VzdalenostiOkolo** (ř. 10, metoda viz níže), která naplní matici vzdáleností vzdálenostmi jednotlivých buněk od té výchozí. Následně již stačí pouze nalézt maximum (tj. největší vzdálenost) v této matici vzdáleností (ř. 12–16) a jeho souřadnice (čili souřadnice nejvzdálenější buňky) a tyto hodnoty vrátit.

```
1: static void VzdalenostiOkolo(int posY, int posX, int[,] mapa,
                               int[,] vzdalenosti)
2: {
3:     var s = (Smer)mapa[posY, posX]; // Načtení hodnoty buňky
4:     foreach (var smer in Smery)
5:     {
6:         if (!s.HasFlag(smer) || // Přepážka uzavřena, nebo...
7:             !Posun(mapa, posY, posX, smer, out int newY,
8:                   out int newX) || // Vede z mapy? (+ posun)...
9:             vzdalenosti[newY, newX] >= 0 && // nebo už tam byl...
10:            vzdalenosti[newY, newX] <= vzdalenosti[posY, posX]
11:                + 1) // ... kratší cestou
12:             continue; // Ve všech těchto případech TUDY NE
13:            // Nastavení nové vzdálenosti pro cílovou buňku
14:            vzdalenosti[newY, newX] = vzdalenosti[posY, posX] + 1;
15:            // Předat rekurzivní zpracování cílové buňce
16:            VzdalenostiOkolo(newY, newX, mapa, vzdalenosti);
17:     }
18: }
```

Procházení bludiště metodou **VzdalenostiOkolo** probíhá rekurzivně, podobně, jako při jeho generování. Rozdíl je pouze v tom, že se mapa bludiště již nijak nemění, a naopak se její přepážky berou jako nepřekročitelné hranice.

Pro každý směr, které již tentokrát nemusí být v náhodném pořadí, jsou kontrolována sousední políčka toho aktuálně zpracovávaného (ř. 4). Ověřuje se tedy, zda je přepážka v daném směru otevřena (ř. 6) a pokud ano, tak rovnou v rámci tohoto ověřování je vyzkoušen přesun na sousední políčko tímto směrem pomocí metody **Posun** (ř. 7). Jestliže se posun zdaří, získáváme rovnou i souřadnice této sousední buňky (**newY** a **newX**), a můžeme tak otestovat, jestli už náhodou dané políčko nebylo zpracováno dříve (např. vstupem z jiné strany okolo smyčky; ř. 8) a pokud ano, tak porovnáme hodnoty vzdáleností obou přístupů (toho dřívějšího a současného; ř. 9), přičemž zvítězí ten kratší.

Jsou-li všechny tyto podmínky splněny, nastavíme vzdálenost pro toto sousední pole o jednu vyšší, než byla vzdálenost na políčko, ze kterého jsme na toto přišli (ř. 12) a metoda rekurzivním zavoláním sebe sama předá zpracování na toto další pole (ř. 14).

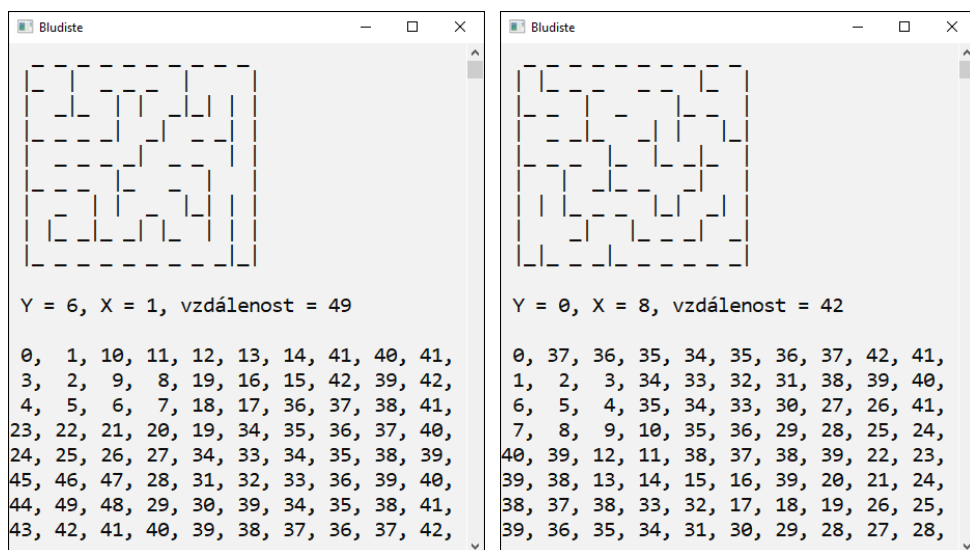
Tímto je tedy hledání nejvzdálenější buňky v bludišti hotové a můžeme vše vyzkoušet v hlavní metodě programu **Main**.

```
1: static void Main(string[] args)
2: {
3:     var mapa = Bludiste.Vytvor(8, 10); // Vytvoření mapy
4:     Bludiste.PridejSmycky(mapa);       // Přidání smyček
5:     int vzdalenost = Bludiste.NejvzdalenejsiPole(0, 0, mapa,
        out int konecY, out int konecX); // Nejvzdálenější pole
6:     Bludiste.Vypis(mapa);              // Vypsání mapy
7:     Console.WriteLine($"Y = {konecY}, X = {konecX}, " +
        $"vzdálenost = {vzdalenost}"); // Výp. souř. nejvz. pole
8:     Console.ReadLine();                // Vyčkání na Enter
9: }
```

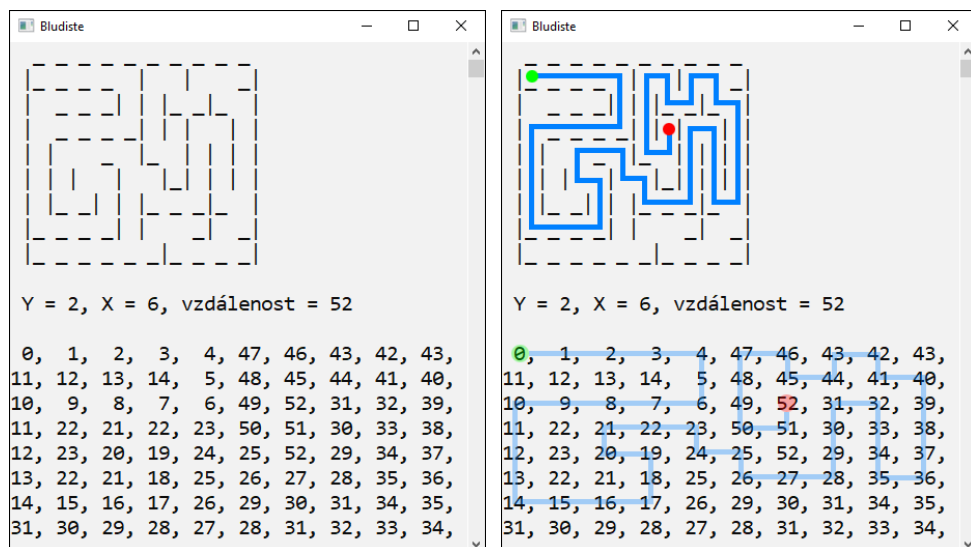
Zde nejprve opět vygenerujeme mapu bludiště (ř. 3), můžeme přidat smyčky (ř. 4) a následně necháme bludiště prohledat z výchozích souřadnic

[0, 0] (samozřejmě lze prohledávání spustit i z jakýchkoli jiných souřadnic) pro nalezení nejvzdálenější buňky (ř. 5). Vygenerované bludiště si můžeme vypsát (ř. 6), stejně jakož i souřadnice a vzdálenost nejvzdálenější buňky (ř. 7).

Pokud do tohoto výpisu přidáme i výpis matice vzdáleností (na ukázkou výstupu bylo zařazeno do metody **NejvzdalenejsiPole** volání dříve přestavené metody **VypisMatici**, viz kap. 2.2), pak může takový výstup vypadat například následovně (výstupy dvou různých spuštění programu pro rozměr 8x10).



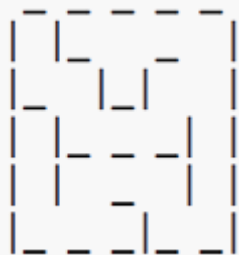
V následující ukázce je 2x totéž bludiště, přičemž ve druhém obrázku je pro názornost modře zvýrazněna cesta od zeleného startu k červenému cíli, jak v přepážkové mapě bludiště, tak i ve výpisu matice vzdáleností.



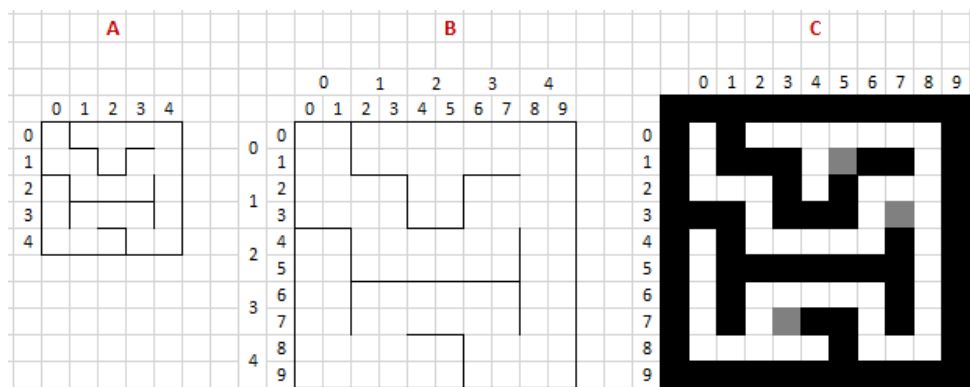
### 3.6 Konverze na blokové bludiště

Vytvořená mapa je tedy bludištěm s přepážkami. To znamená, že na každé jeho políčko lze někudy vstoupit a to, mezi kterými dvěma políčky lze či nelze přejít určují přepážky, které mají vzhledem k políčkům nulovou tloušťku. Toto se dobře vykresluje čarami ve čtvercové síti (gridu / tabulce), nicméně pokud má mít hra, pro kterou je bludiště určeno, nějaký prostorový charakter, ať již 2D nebo i 3D, je třeba z přepážek udělat samostatné prostorové čtvercové bloky, o velikosti samotných políček. Postup této konverze si ukážeme na bludišti o rozměrech 5x5, jehož kód a vzhled mapy je následující.

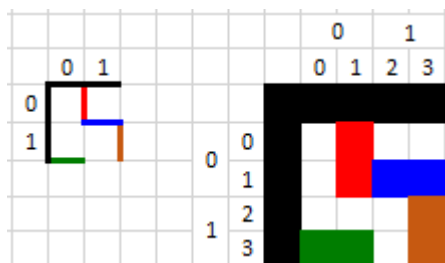
```
1: var mapa = new int[,] {
2:     {2, 4, 2+4+8, 4+8, 2+8 },
3:     {1+4, 2+8, 1, 2+4, 1+2+8 },
4:     {2, 1+4, 4+8, 1+8, 1+2 },
5:     {1+2, 2+4, 4+8, 2+8, 1+2 },
6:     {1+4, 1+4+8, 8, 1+4, 1+8 }
7: };
```



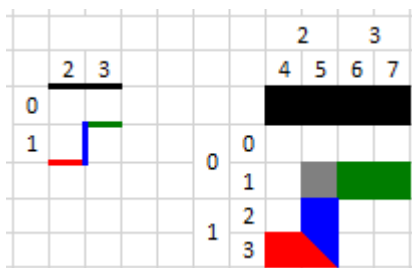
Celý postup konverze si rozkresleme do několika kroků. Jelikož se přepážky **(A)** mají stát prostorovými bloky, je třeba rozměry bludiště nejprve zdvojnásobit **(B)**. V dalším kroku pak pro každé původní políčko, kde byla přepážka vpravo nebo dole, přidáme blok zdi do nového dvojbloku k jeho pravé případně k dolní straně **(C)**.



Jak se která přepážka promítne do kterých dvou bloků zdí, ukazuje následující barevně vyznačený obrázek.



V určitých případech však dochází k tomu, že blok zdi není dosazen až do rohů a vzniká tak diagonálně průchozí políčka. To ukazuje následující výřez konverze.



Políčko s původními souřadnicemi [1, 2] (souřadnice jsou uváděny jako index řádku a index sloupce, jako v maticích) má přepážku dole i vpravo, tak je přidán blok zdi na nové souřadnice [3, 4] a [3, 5] (za dolní přepážku), a na [2, 5] a [3, 5] (za pravou přepážku). Na nové souřadnici [3, 5] je tak přidána zeď  $2x$ , což výsledek nijak negativně neovlivní.

Nicméně když někde zdi přebývají, je logické, že jinde by mohly pro změnu chybět. Políčko s původními souřadnicemi [0, 2] nemá dole ani vpravo žádnou přepážku, ta tudíž ani nepřidá žádný blok zdi. Na výsledném políčku s novými souřadnicemi [1, 5] se tak zeď nepřidá a vznikne nevzhledný, diagonálně průchozí, „zub“.

Tento problém však nastává pouze výjimečně na polích s oběma lichými indexy a chybějící zeď lze identifikovat z přítomnosti zdi pod nebo napravo od tohoto políčka v nové mapě. Díky tomu není problém na nové mapě provést korekci, která problém opraví. Celý kód konverze, včetně této korekce uvádí následující metoda **Konverze**.





```
1: /// <summary>Konverze přepážkového bludiště na blokové</summary>
2: /// <returns>Mapa blokového bludiště, kde false = průchozí blok,
3: /// true = neprůchozí blok (zeď)</returns>
4: public static bool[,] Konverze(int[,] mapaP)
5: {
6:     // Deklarace 2D pole pro blokovou mapu dvojité velikosti
7:     var mapaB = new bool[mapaP.GetLength(0) * 2,
8:                          mapaP.GetLength(1) * 2];
9:     // Konverze bludiště
10:    for (int i = 0; i < mapaP.GetLength(0); i++)
11:        for (int j = 0; j < mapaP.GetLength(1); j++)
12:        {
13:            var s = (Smer)mapaP[i, j];
14:            if (!s.HasFlag(Smer.P))
15:            {
16:                mapaB[i * 2, j * 2 + 1] = true;
17:                mapaB[i * 2 + 1, j * 2 + 1] = true;
18:            }
19:            if (!s.HasFlag(Smer.D))
20:            {
21:                mapaB[i * 2 + 1, j * 2] = true;
22:                mapaB[i * 2 + 1, j * 2 + 1] = true;
23:            }
24:        }
25:    // Korekce (doplnění) lichých bloků
26:    for (int i = 1; i < mapaB.GetLength(0) - 1; i += 2)
27:        for (int j = 1; j < mapaB.GetLength(1) - 1; j += 2)
28:            if (mapaB[i, j + 1] || mapaB[i + 1, j])
29:                mapaB[i, j] = true;
30:    return mapaB;
}
```

Výstupem konverze je opět matice, tentokrát ovšem nikoli číselná, ale logických hodnot (**bool[,]**). Logická hodnota **false** v ní značí průchozí blok, logická hodnota **true** pak neprůchozí blok, tedy zeď.



### 3.7 Vykreslování blokové mapy do konzole

Nyní přidáme metodu, která tuto novou blokovou mapu bludiště vypíše do konzolového okna. Pro průchozí pole je zde již pouze mezera a pro blok se zdí je tentokrát použit znak vyplněného obdélníku, který lze nalézt přes mapu znaků. Výpis samotný je pak jednodušší než u verze s přepážkami, protože jedno políčko je vždy právě jeden znak.

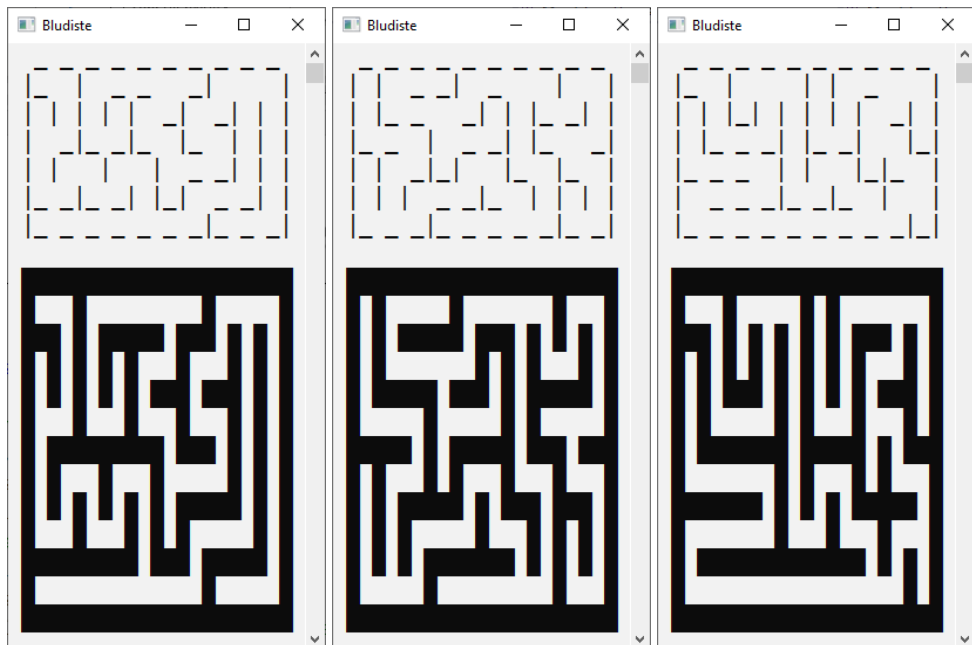
```
1: /// <summary>Vypsání blokového bludiště do konzole</summary>
2: public static void Vypis(bool[, ] mapa)
3: {
4:     // Horní hranice bludiště
5:     Console.WriteLine(" ".PadRight(mapa.GetLength(1) + 2, '█'));
6:     // Řádky bludiště
7:     for (int i = 0; i < mapa.GetLength(0); i++)
8:     {
9:         Console.Write("█"); // Levá hranice bludiště
10:        // Sloupce v řádku (buňky)
11:        for (int j = 0; j < mapa.GetLength(1); j++)
12:            Console.Write(mapa[i, j] ? "█" : " ");
13:        Console.WriteLine();
14:    }
15: }
```

Následně můžeme použít znovu všechny metody v hlavní části programu.

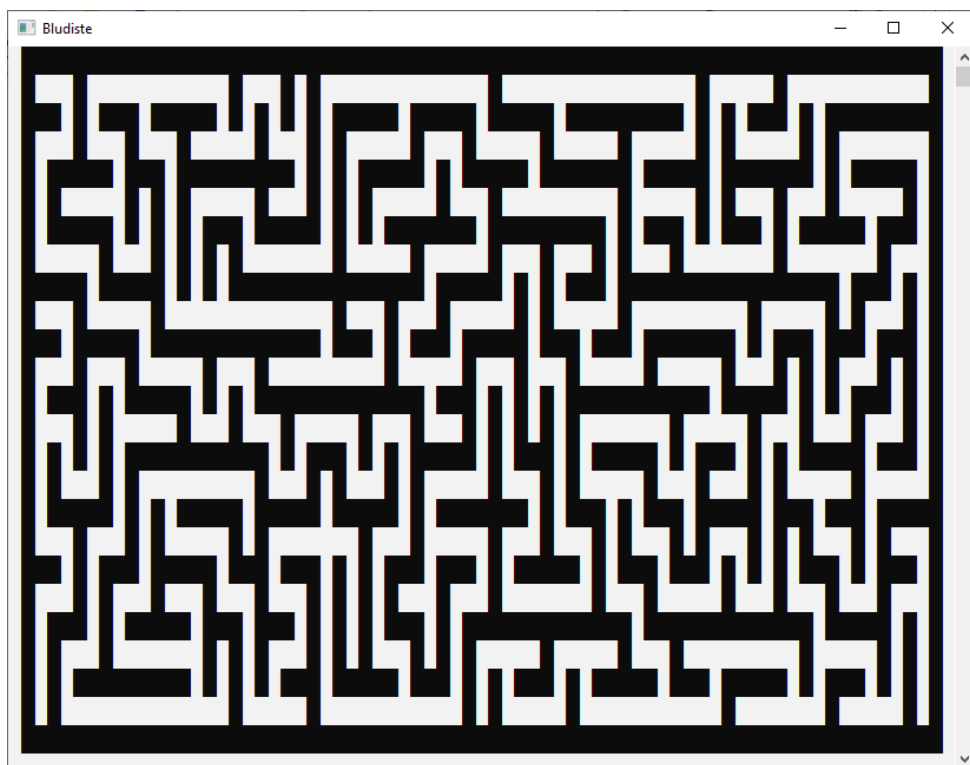


```
1: static void Main(string[] args)
2: {
3:     // Vygenerování a vykreslení přepážkové mapy
4:     var mapa = Bludiste.Vytvor(6, 10);
5:     // Bludiste.PridejSmycky(mapa); // Možnost přidat smyčky
6:     Bludiste.Vypis(mapa);
7:     Console.WriteLine();
8:     // Konverze na blokovou mapu a její vykreslení
9:     var mapaB = Bludiste.Konverze(mapa);
10:    Bludiste.Vypis(mapaB);
11:    Console.ReadLine();           // Vyčkání na stisk Enteru
12: }
```

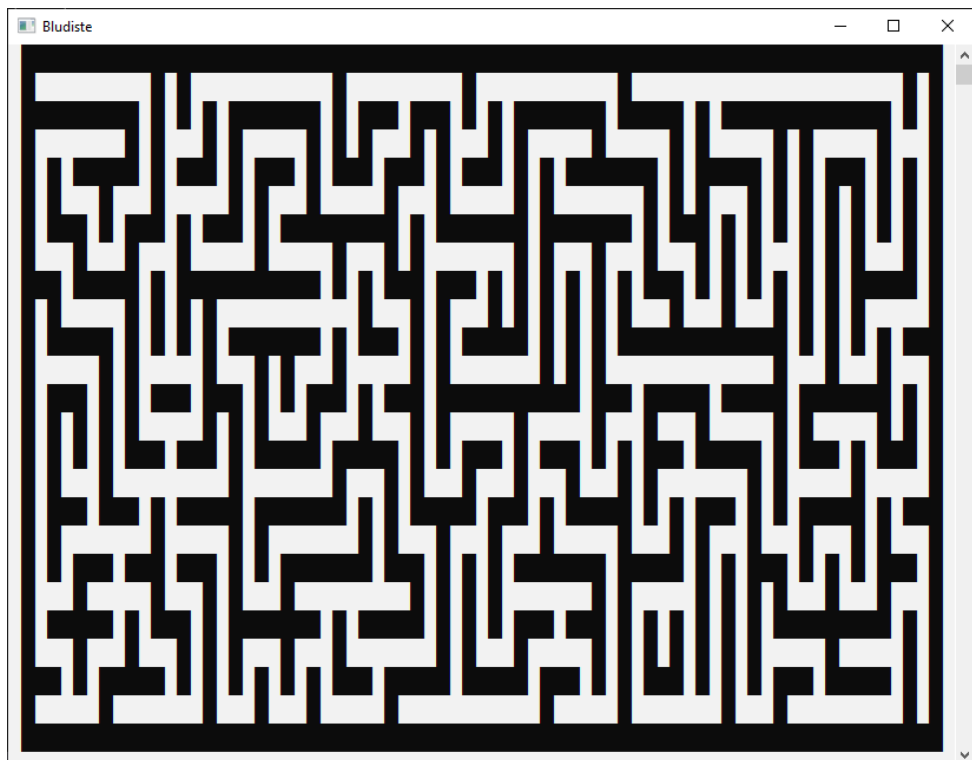
Výsledné výstupy pak mohou vypadat například následovně (3 různá spuštění pro rozměr 6x10).



Jak je z výstupů patrné, na šířku jsou mapy ve výpisu stejně velké, protože přepážková verze používala pro výpis každé buňky dva znaky na šířku. Bloková verze je ovšem dvojnásobná i na výšku, kde u přepážek stačil pouze jeden znak. Ve větším rozměru (12x35) pak samostatná bloková mapa může vypadat například následovně.



A při totéž rozměru (12x35) jiná mapa, avšak s přidáním 5 % smyček (tj. 21), pak může vypadat výpis blokové mapy následovně.





Tento materiál vznikl v rámci realizace projektu  
Strategický rozvoj Univerzity Hradec Králové, reg. č. CZ.02.2.69/0.0/0.0/16\_015/0002427



EVROPSKÁ UNIE  
Evropské strukturální a investiční fondy  
Operační program Výzkum, vývoj a vzdělávání



MINISTERSTVO ŠKOLSTVÍ,  
MLÁDEŽE A TĚLOVÝCHOVY