Microsoft

Ján Hanák

C++/CLI – Praktické príklady

Príručka praktických cvičení pre vývojárov, programátorov a softvérových expertov

Microsoft Visual Studio

Microsoft .NET

Ján Hanák

C++/CLI – Praktické príklady

(Príručka praktických cvičení pre vývojárov, programátorov
a softvérových expertov)

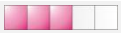
Artax
2009

Autor: Ing. Ján Hanák, MVP

C++/CLI – Praktické príklady

Vydanie:	prvé
Rok prvého vydania:	2009
Náklad:	150 ks
Jazyková korektúra:	Ing. Peter Kubica
Vydal:	Artax a.s., Žabovřeská 16, 616 00 Brno pre Microsoft s.r.o., Vyskočilova 1461/2a, 140 00 Praha 4
Tlač:	Artax a.s., Žabovřeská 16, 616 00 Brno
ISBN:	978-80-87017-05-0

C++/CLI – Praktické príklady

Vedomostná náročnosť: 

Cieľové publikum: **mierne pokročilí** a **pokročilí** vývojári v jazyku **C++/CLI**

Časová náročnosť: **1** hodina **55** minút

Programovacie jazyky: **C++/CLI**, **C# 4.0**

Vývojové prostredia: **Visual Studio 2010** (Beta 1)

Operačné systémy: **Windows Vista**, Windows 7, Windows XP

Technológie: **BCL** platformy **.NET Framework 4.0**

Programovací jazyk C++/CLI bol uvedený dovedna so softvérovým produktom Visual Studio 2005 spoločnosťou Microsoft v roku 2005. Odvtedy sa tomuto jazyku podarilo získať priazeň početného segmentu C++ vývojárov, ktorí hľadali adekvátny ekvivalent svojho obľúbeného „inkrementovaného céčka“ pre vývoj riadených aplikácií na platforme .NET Framework. Riadené C++, ako C++/CLI často radi nazývame, ponúka „dotnet“ programátorom veľa konkurenčných výhod, ktoré prídu vhod najmä pri praktickom programovaní robustných aplikácií. Intuitívne syntaktické konštrukcie, košatá jazyková špecifikácia, snaha o maximálne znovupoužitie elementov jazyka C++ či začlenenie automatického správcu pamäte – to je milý štvorlístok noviniek, ktoré potešia každého skutočného vývojára.

Keďže základný výučbový kurz algoritmizácie a programovania v jazyku C++/CLI sme podali vo vysokoškolskej učebnici *C++/CLI – Začíname programovať*[1], v tejto príručke praktických cvičení vás zoznámime s tvorbou objektovo orientovaných programov v jazyku C++/CLI. Pri praktickom programovaní budeme využívať vývojové prostredie Microsoft Visual Studio 2010, ktoré sa v čase tvorby tohto diela nachádzalo v štádiu prvej betaverzie (Beta 1).

Aby ste mohli z príručky praktických cvičení vyťažiť maximum, predpokladáme, že ovládate základy programovania v jazyku C++/CLI. Nebudeme sa teda zaoberať vysvetľovaním elementárnych princípov a programovacích elementov, ako sú premenné, operátory, rozhodovacie príkazy či cykly.

[1] Hanák, J.: *C++/CLI – Začíname programovať*. Brno: Artax, 2009.
Vysokoškolskú učebnicu si môžete v elektronickej forme zadarmo prevziať z nasledujúcej adresy: http://msdn.microsoft.com/cs-cz/dd727769.aspx.

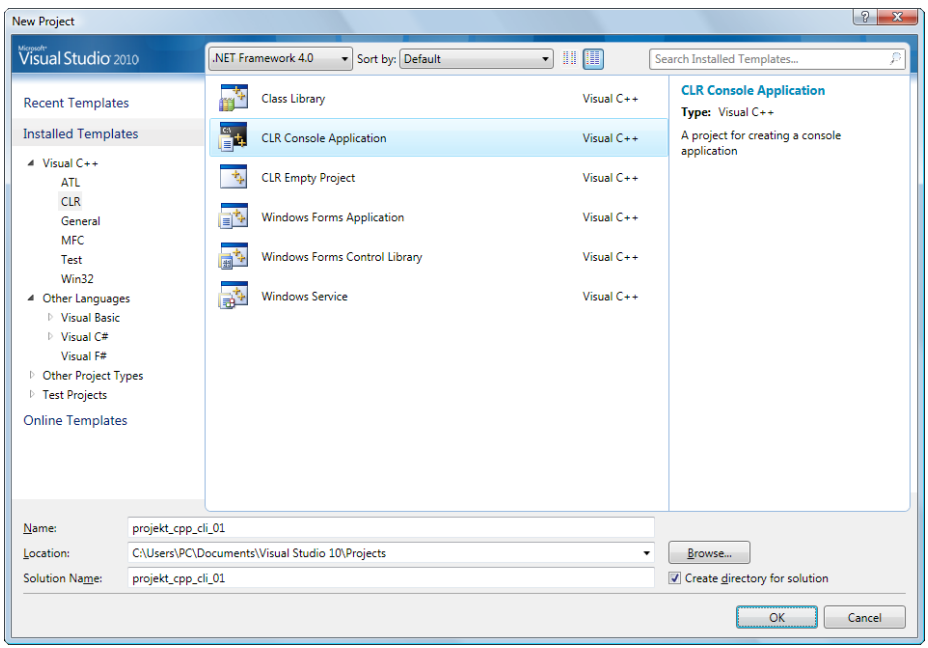
Rovnako počítame s tým, že ste absolvovali základný kurz objektovo orientovaného programovania v jazyku C++/CLI. Naším cieľom je využiť vašu existujúcu bázu znalostí pri budovaní prakticky orientovaných programov v jazyku C++/CLI.

Obsah

1 Založenie nového projektu štandardnej konzolovej aplikácie jazyka C++/CLI vo vývojovom prostredí Visual Studio 2010

Nový projekt štandardnej konzolovej aplikácie jazyka C++/CLI založíme vo vývojovom prostredí Visual Studio 2010 nasledujúcim spôsobom:

1. Na úvodnej stránke **Start Page** klikneme na položku **Projects**, a potom aktivujeme položku **New Project**.
2. V dialógovom okne **New Project** sa zameriame na stromovú štruktúru **Installed Templates**, z ktorej vyberieme položku **Visual C++ → CLR**.
3. Zo zoznamu projektových šablón zvolíme šablónu **CLR Console Application**.
4. Do textového poľa **Name** zadáme názov projektu. Visual C++ 2010 automaticky vyplní aj textové pole **Solution Name**, a to tak, aby malo riešenie rovnaký názov ako projekt, ktorý bude v riešení uložený. V tejto chvíli by malo dialógové okno **New Project** vyzerať ako na obr. 1.



Obr. 1: Založenie nového projektu štandardnej konzolovej aplikácie jazyka C++/CLI

5. Klikneme na tlačidlo **OK**.

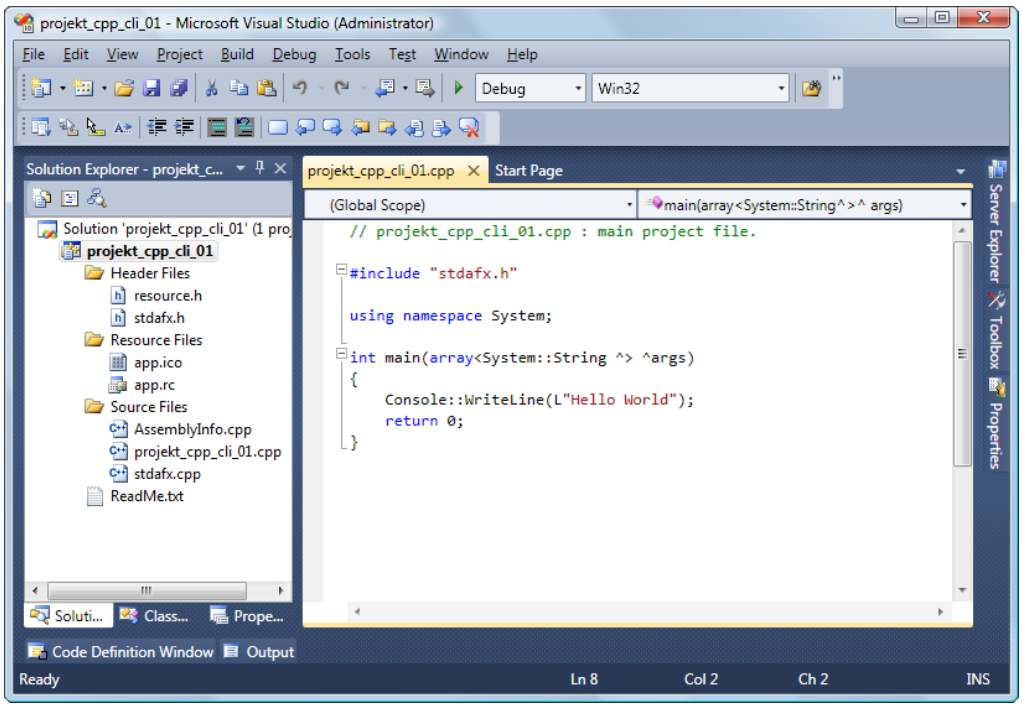
6. Spustí sa sprievodca založením projektu štandardnej konzolovej aplikácie **CLR Console Application**. Len čo sprievodca vytvorí riešenie a projekt, pridá do projektu všetky potrebné súčasti a otvorí hlavný zdrojový súbor aplikácie v editore zdrojového kódu vývojového prostredia Visual Studio 2010. Ak nie je stanovené inak, tak riešenie s projektom bude uložené do nasledujúcej lokality: C:\Users\<MenoPoužívateľa>\Documents\Visual Studio 10\Projects. Hlavný zdrojový súbor aplikácie sa volá rovnako ako projekt, ktorého názov sme zadali v dialógovom okne **New Project**. Koncovka hlavného zdrojového súboru je .cpp, podobne ako pri zdrojových súboroch natívneho C++. Ako si teda môžeme všimnúť, tak len podľa koncovky zdrojového súboru nevieme povedať, či sa v ňom nachádza zdrojový kód jazyka C++ alebo C++/CLI.

   Ak je vývojové prostredie nakonfigurované podľa profilu vývojára vo Visual C++ (čo predpokladáme), tak internú štruktúru riešenia a projektu môžeme vidieť v podokne **Solution Explorer**. Toto podokno je štandardne uložené pozdĺž ľavej strany hlavného okna vývojového prostredia Visual Studio 2010. Prostredníctvom podokna **Solution Explorer** vidíme, že v projekte sú celkovo tri typy aplikačných súborov. Tieto sú roztriedené do priečinkov **Header Files** (Hlavičkové súbory), **Resource Files** (Súbory s aplikačnými zdrojmi) a **Source Files** (Zdrojové súbory).

Poznámka: Je možné, že vzhľad a kompozícia vášho vývojového prostredia je iná, alebo je možno zvolený iný vývojársky profil (napr. pre jazyk C# či Visual Basic). Aby ste mohli hladko sledovať ďalší výklad, odporúčame vám uprednostniť vývojársky profil pre jazyk C++. Konfiguráciu vývojového prostredia pre potreby C++ programátorov uskutočníte takto:

- Z ponuky **Tools** vyberiete príkaz **Import and Export Settings**.
- V 1. kroku sprievodcu vyberiete voľbu **Reset all settings** a kliknete na tlačidlo **Next**.
- V 2. kroku sprievodcu kliknete na voľbu **No, just reset settings, overwriting my current settings** a vzápätí aktivujete tlačidlo **Next**.
- V 3. kroku sprievodcu označíte vývojársky profil **Visual C++ Development Settings** a stlačíte tlačidlo **Finish**.

Okrem všetkého, čo sme zatiaľ spomenuli, musíme dodať, že sprievodca vygeneruje aj implicitný zdrojový kód, ktorý vloží do hlavného zdrojového súboru. Programové príkazy automaticky zostaveného fragmentu zdrojového kódu sú uvedené na obr. 2.



Obr. 2: Hlavný zdrojový súbor s implicitne vygenerovanými programovými príkazmi jazyka C++/CLI

Samočinne zostavený zdrojový kód je ľahko čitateľný:

1. Direktívou predprocesora **#include** je zavedený odkaz na hlavičkový súbor stdafx.h.
2. Direktívou prekladača **using namespace** dochádza k importu všetkých pomenovaných entít z koreňového menného priestoru **System**.
3. Definícia hlavnej metódy **main**, ktorá pôsobí ako vstupný bod riadenej aplikácie jazyka C++/CLI. Zo syntaktického obrazu metódy **main** je zrejmé, že ide o parametrickú metódu (s jedným formálnym parametrom **args**, ktorý je schopný prijať odkaz na vektorové pole obsahujúce textové reťazce – argumenty príkazového riadka) s celočíselnou návratovou hodnotou (jazyk C++/CLI je v tomto

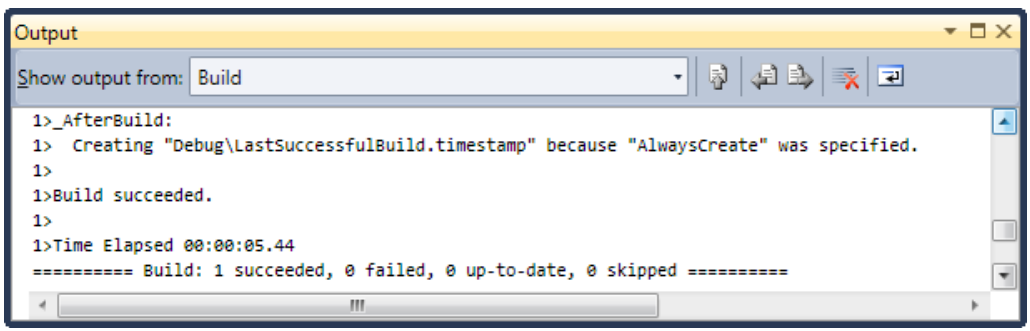
smere konformný s ISO štandardom jazyka C++, takže nulová návratová hodnota predstavuje normálne ukončenie spracovania riadenej aplikácie). V tele hlavnej metódy **main** je situovaný výkonný príkaz na zobrazenie uvítacej správy.

2 Zostavenie štandardnej konzolovej aplikácie jazyka C++/CLI vo vývojovom prostredí Visual Studio 2010

V procese zostavenia programu jazyka C++/CLI sa odohrávajú tieto činnosti: preloženie zdrojového súboru programu prekladačom, jeho prepojenie s exekučným prostredím spojovacím programom (linkerom) a vygenerovanie priamo spustiteľného súboru. Akúkoľvek riadenú aplikáciu jazyka C++/CLI zostavíme vo vývojovom prostredí Visual Studio 2010 týmto spôsobom:

1. Otvoríme ponuku **Build** a aktivujeme príkaz **Build Solution** alebo príkaz **Build <NázovProjektu>**. Vzhľadom na to, že pracujeme s jednoprojektovým riešením, je jedno, ktorý príkaz uprednostníme. (S výhodou môžeme využiť aj klávesové skratky **F7** či **CTRL+SHIFT+B**).

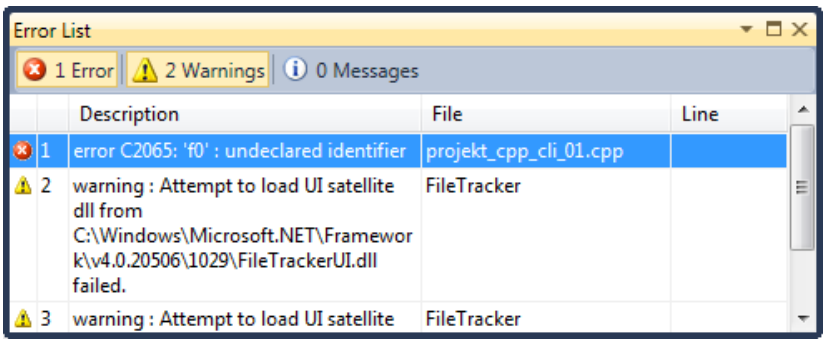
2. Proces zostavenia programu môžeme sledovať v podokne **Output**. Ak je všetko v poriadku, v podokne **Output** sa nakoniec zobrazí správa *Build: 1 succeeded* (obr. 3).



Obr. 3: Podokno **Output** dokumentuje informačné správy, ktoré sprevádzajú proces zostavenia riadenej aplikácie jazyka C++/CLI

2.1 Diagnostika a korekcia syntakticko-sémantických programových chýb

Ak v procese zostavovania aplikácie zahlási Visual C++ 2010 chybu, tak zobrazíme podokno **Error List** (**View** → **Other Windows** → **Error List**). V podokne **Error List** aktivujeme zobrazovanie chýb (**Errors**), varovaní (**Warnings**) a informačných správ (**Messages**). Aktivácia v tomto kontexte znamená kliknutie na tlačidlá s príslušnými textovými identifikátormi. Vzápätí sa zobrazia chyby, ktoré prekladač detegoval. Presný výskyt chyby zistíme tak, že dvakrát klikneme na jej opis v podokne **Error List**. Editor zdrojového kódu nás prenesie na ten riadok zdrojového kódu aplikácie, na ktorom sa chyba nachádza. Len čo chybu opravíme, uskutočníme opätovné zostavenie aplikácie.



Obr. 4: Podokno **Error List** nám umožňuje rýchlo nájsť syntakticko-sémantické programové chyby

3 Spustenie zostavenej štandardnej aplikácie jazyka C++/CLI z vývojového prostredia Visual Studio 2010

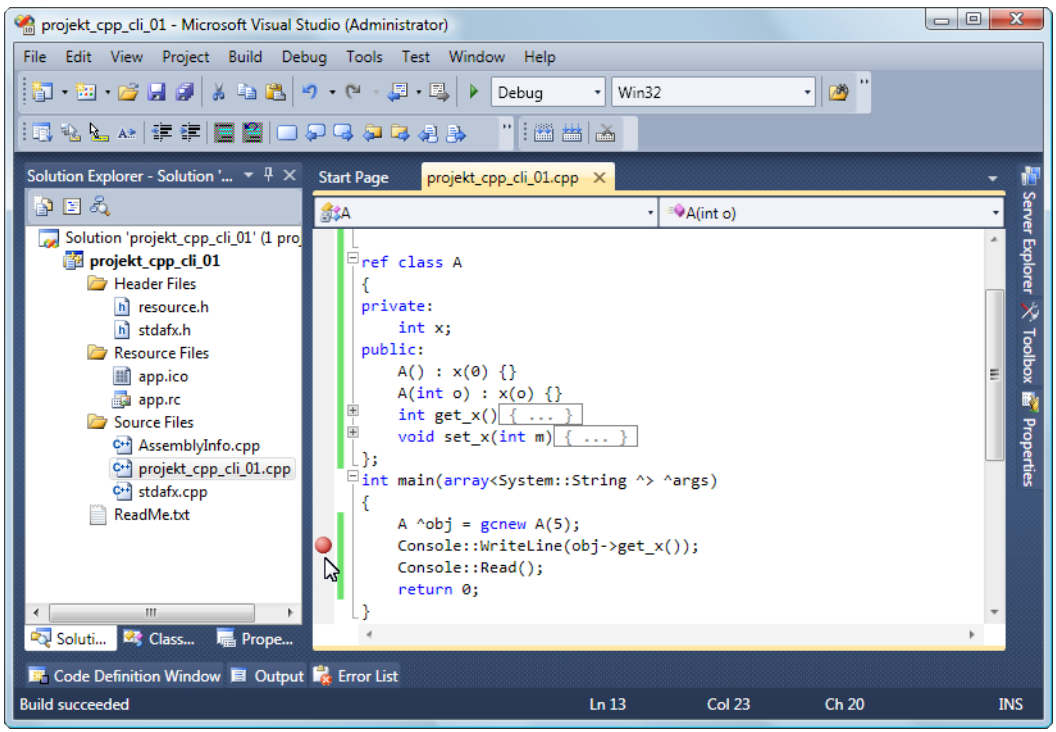
Zostavenú riadenú aplikáciu jazyka C++/CLI spustíme nasledujúcim spôsobom: otvoríme ponuku **Debug** a vyberieme položku **Start Debugging** (alebo rýchlejšie použijeme klávesovú skratku **F5**). Aplikácia sa spustí, pričom máme možnosť preveriť, či produkuje očakávané výstupy.

4 Vloženie lokálneho bodu prerušenia a monitorovanie dátových entít programu jazyka C++/CLI

Lokálny bod prerušenia (angl. *breakpoint*) slúži na pozastavenie spracovania programu a na diagnostiku aktuálnych stavov dátových entít programu.

Lokálny bod prerušenia spojíme s riadkom zdrojového kódu programu jazyka C++/CLI takto:

1. Kurzor myši umiestnime na požadovaný riadok zdrojového kódu.
2. Ľavým tlačidlom myši klikneme na sivý vertikálny pruh, ktorý sa nachádza naľavo od zdrojového kódu. (Ekvivalentne môžeme upotrebiť klávesovú skratku **F9**.)
3. Ak je všetko v poriadku, tak sa objaví červená guľôčka (●), ktorá indikuje prítomnosť lokálneho bodu prerušenia (obr. 5).

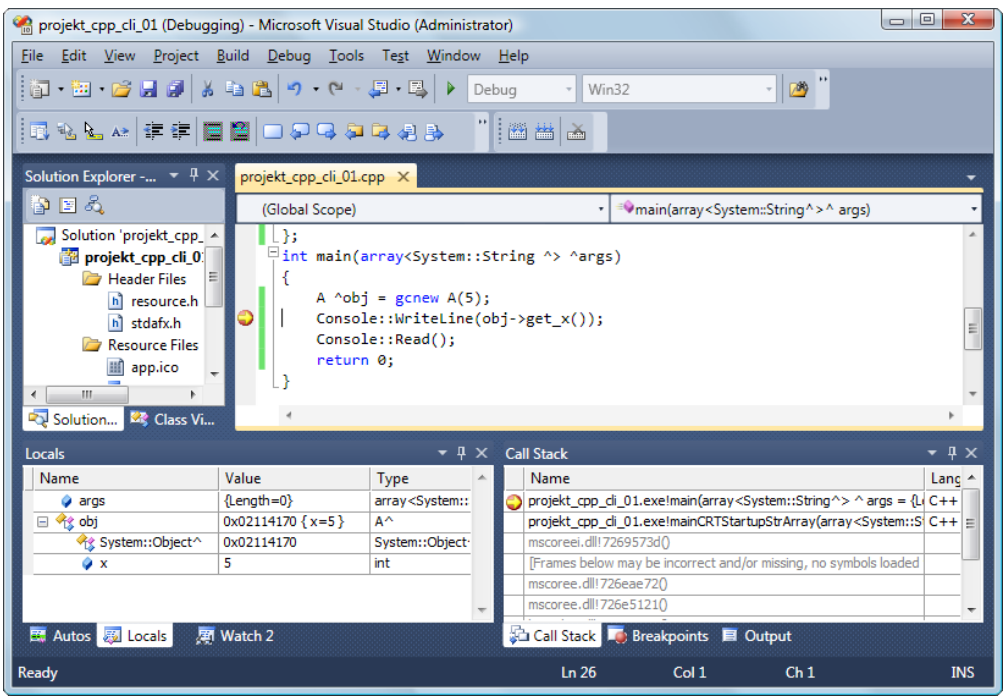


Obr. 5: Vloženie lokálneho bodu prerušenia

4. Keď chceme pomocou lokálneho bodu prerušenia sledovať, aké obsahy majú dátové entity, ktoré v našom programe používame, tak musíme program po zostavení spustiť príkazom **Debug** → **Start Debugging** (alebo pomocou klávesovej skratky **F5**).

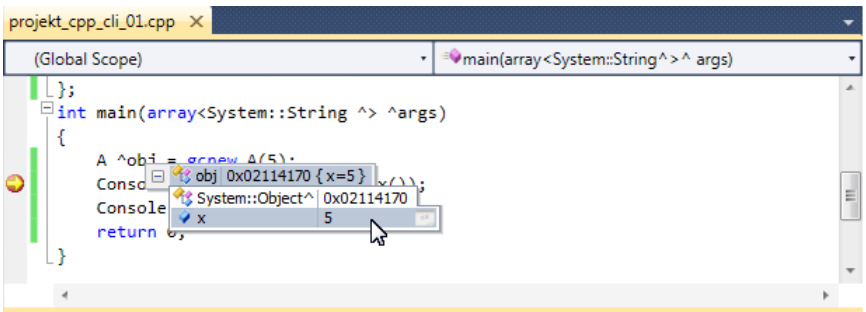
5. Po dosiahnutí lokálneho bodu prerušenia sa program z režimu spracovania prenesie do režimu prerušenia. Dialógové okno, v ktorom program beží, sa dostane do pozadia. Naopak, v popredí sa ocitne okno vývojového prostredia Visual Studio 2010. Riadok, s ktorým je lokálny bod prerušenia zviazaný, bude vyznačený červenou guľôčkou a žltou šípkou. Žltá šípka ukazuje na príkaz, ktorý bude spracovaný ako prvý vtedy, keď dôjde k obnoveniu behu programu.

6. Obsahy požadovaných dátových entít (napr. lokálnych premenných, objektov, polí či inštancií štruktúr) môžeme monitorovať pomocou podokien **Autos** a **Locals**, ktoré sa implicitne zobrazia (obr. 6).



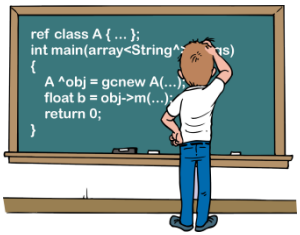
Obr. 6: Aplikácia jazyka C++/CLI sa nachádza v režime prerušenia svojho spracovania, pretože bol detegovaný lokálny bod prerušenia

Ešte rýchlejším riešením je použitie dátových vizualizérov: stačí, aby sme kurzor myši umiestnili nad identifikátor dátovej entity a v príslušnom dátovom vizualizéri sa objaví jej obsah (obr. 7).



Obr. 7: Monitorovanie dátovej sekcie objektu pomocou dátového vizualizéra

7. Beh programu obnovíme voľbou **Debug** → **Continue**, alebo stlačením klávesu **F5**.



5 Praktický príklad č. 1: Program na výpočet smerodajnej odchýlky výberového štatistického súboru dát

Vedomostná náročnosť:

Časová náročnosť: **20** minút

Prvý program, ktorý spoločne vytvoríme, bude automatizovať výpočet smerodajnej odchýlky výberového štatistického súboru dát. Smerodajná odchýlka (niekedy tiež nazývaná ako štandardná odchýlka) je miera variability, ktorá determinuje rozloženie hodnôt skúmaného štatistického znaku vo výberovom súbore dát (teda v tzv. vzorke dát). Pre potreby ďalšej algoritmizácie prijmeme dohovor, že smerodajnú odchýlku budeme počítať z výberového súboru diskrétnych dát. Podľa rigoróznej definície je smerodajná odchýlka druhou odmocninou zo súčtu štvorcov odchýlok hodnôt štatistického znaku od jednoduchého aritmetického priemeru delená $n - 1$ stupňami voľnosti. Formálny matematický vzťah na výpočet smerodajnej odchýlky výberového súboru dát je nasledujúci:

$$s = \sqrt{\frac{\sum_{i=1}^{n}(x_i - \bar{x})^2}{n-1}}$$

kde:

- s je smerodajná odchýlka výberového súboru dát.
- n je rozsah výberového súboru dát.
- x_i je hodnota i-tého štatistického znaku výberového súboru dát.
- $\bar{x}$ je jednoduchý aritmetický priemer výberového súboru dát.

Smerodajná odchýlka slúži na zistenie miery variability výberového súboru dát. Presnejšie povedané, smerodajná odchýlka nám vraví, ako sa v priemere odchyľujú hodnoty štatistického znaku výberového súboru od jednoduchého aritmetického priemeru. Smerodajná odchýlka je malá vtedy, ak sú hodnoty skúmaného štatistického znaku sústredené okolo jednoduchého aritmetického priemeru. Za týchto okolností môžeme konštatovať, že miera variability výberového súboru dát je nízka. Na druhej strane, niekedy sa môžu vo výberovom súbore dát nachádzať významné extrémalne hodnoty, ktoré spôsobujú nárast miery variability, a teda samozrejme aj nárast smerodajnej odchýlky.

Praktický príklad: Uvažujme nasledujúcu modelovú situáciu, v ktorej budeme sledovať dĺžku hospitalizácie pacientov chirurgického oddelenia Fakultnej nemocnice Staré Mesto v Bratislave. Náš výberový súbor dát bude tvoriť 10 pacientov, ktorí v nemocnici strávili rozlične dlhý čas (tab. 1).

Tab. 1: Dĺžka pobytu pacientov na chirurgickom oddelení

P. č. pacienta	Dĺžka hospitalizácie (v dňoch)
1.	5
2.	4
3.	8
4.	12
5.	2
6.	9
7.	6
8.	4
9.	8
10.	5

Ďalej budeme postupovať podľa nasledujúceho algoritmu:

1. Vypočítame jednoduchý aritmetický priemer ($\bar{x}$). Všeobecný matematický vzorec na výpočet $\bar{x}$ má túto podobu:

$$\bar{x} = \frac{\sum_{i=1}^{n} x_i}{n}$$

kde:

- $\bar{x}$ je jednoduchý aritmetický priemer výberového súboru dát.
- n je rozsah výberového súboru dát.

- x_i je hodnota i-tého štatistického znaku výberového súboru dát.

Konkrétna aplikácia v našom modeli:

$$\bar{x} = \frac{63}{10} = 6{,}3$$

Jednoduchý aritmetický priemer má hodnotu 6,3 dní, čo znamená, že v priemere bol každý pacient hospitalizovaný takmer týždeň.

2. Vypočítame štvorce odchýlok hodnôt štatistického znaku od jednoduchého aritmetického priemeru $(x_i - \bar{x})^2$:

Tab. 2: Výpočet štvorcov odchýlok hodnôt od jednoduchého aritmetického priemeru

P. č. pacienta	x_i	$\bar{x}$	$x_i - \bar{x}$	$(x_i - \bar{x})^2$
1.	5	6,3	1,3	1,69
2.	4	6,3	2,3	5,29
3.	8	6,3	-1,7	2,89
4.	12	6,3	-5,7	32,49
5.	2	6,3	4,3	18,49
6.	9	6,3	-2,7	7,29
7.	6	6,3	0,3	0,09
8.	4	6,3	2,3	5,29
9.	8	6,3	-1,7	2,89
10.	5	6,3	1,3	1,69

3. Vypočítame smerodajnú odchýlku:

$$s = \sqrt{\frac{\sum_{i=1}^{n}(x_i - \bar{x})^2}{n-1}} = \sqrt{\frac{78{,}1}{9}} = 2{,}95$$

Smerodajná odchýlka má hodnotu 2,95 dní, čo znamená, že v priemere sa dĺžka hospitalizácie pacientov odchyľuje od ich priemernej dĺžky pobytu v nemocnici o takmer 3 dni.

Praktická algoritmizácia programu:

```
#include "stdafx.h"

using namespace System;

int main(array<System::String ^> ^args)
{
    // Definície premenných.
    const int pocetPacientov = 10;
    array<int> ^pobytyPacientov = gcnew array<int>(10);
    array<float> ^stvorceOdchylok = gcnew array<float>(10);
    int sucetPobytov = 0;
    float sucetStvorcovOdchylok = 0.0f;
    float priemer, smerodajnaOdchylka;
    // Získanie hodnôt výberového súboru dát.
    for(int i = 0; i < pocetPacientov; i++)
    {
        Console::Write("Zadajte dĺžku pobytu {0}. pacienta: ", i + 1);
        pobytyPacientov[i] = Convert::ToInt32(Console::ReadLine());
        sucetPobytov += pobytyPacientov[i];
    }
    // Výpočet jednoduchého aritmetického priemeru.
    priemer = sucetPobytov / (float)pocetPacientov;
    // Výpočet štvorcov odchýlok a ich súčtu.
    for(int i = 0; i < pocetPacientov; i++)
    {
        stvorceOdchylok[i] = pobytyPacientov[i] - priemer;
        stvorceOdchylok[i] *= stvorceOdchylok[i];
        sucetStvorcovOdchylok += stvorceOdchylok[i];
    }
    // Výpočet smerodajnej odchýlky.
    smerodajnaOdchylka =
        (float)Math::Sqrt(sucetStvorcovOdchylok / (pocetPacientov - 1));
```

```
    // Zobrazenie jednoduchého aritmetického priemeru a smerodajnej odchýlky
    // na výstupe.
    Console::WriteLine("Jednoduchý aritmetický priemer: {0}.", priemer);
    Console::WriteLine("Smerodajná odchýlka: {0}.", smerodajnaOdchylka);
    Console::Read();
    return 0;
}
```

Komentár k zdrojovému kódu: Syntaktický obraz programu jazyka C++/CLI verne kopíruje všeobecný algoritmus na výpočet smerodajnej odchýlky výberového súboru dát. Dĺžku pobytov jednotlivých pacientov v nemocnici ukladáme do vektorového poľa. Podobným štýlom manipulujeme tiež so štvorcami odchýlok od jednoduchého aritmetického priemeru. Ak ste s jednorozmernými poľami v jazyku C++/CLI ešte nepracovali, dovolíme si ich stručne uviesť. Vzhľadom na to, že vytvárame riadenú aplikáciu jazyka C++/CLI, pracujeme s riadenými poľami. Riadené polia sú objekty, ktoré sú alokované v riadenej halde fyzického procesu aplikácie jazyka C++/CLI. Riadené pole vzniká v procese inštanciácie. V uvedenom zdrojovom kóde dochádza k inštanciácii dvoch riadených polí: prvé pracuje s celočíselnými prvkami, zatiaľ čo druhé dokáže uchovať reálne prvky s jednoduchou presnosťou. Pre plynulý tok výkladu uvádzame inštančné príkazy oboch polí znovu:

```
array<int> ^pobytyPacientov = gcnew array<int>(10);
array<float> ^stvorceOdchylok = gcnew array<float>(10);
```

Zámer vytvoriť riadené pole vyjadríme použitím inštanciačného operátora **gcnew**, ktorý aplikujeme na výraz **array<T>(R)**, kde **T** je dátový typ prvkov poľa a **R** je rozsah poľa. Operátor **gcnew** zabezpečí alokáciu poľa v 1. generácii riadenej haldy, vykoná jeho implicitnú inicializáciu a vráti odkaz na vytvorené pole. Tento odkaz je v našom prípade ukladaný do príslušnej odkazovej premennej, ktorej typom je **array<T>^**. Skonštruované jednorozmerné pole predstavuje objekt, no anonymný, a to v tom zmysle, že pole nemá v riadenej halde žiadne symbolické pomenovanie. Ak chceme s poľom narábať, musíme vždy použiť odkazovú premennú, ktorá je inicializovaná odkazom na príslušné pole.

Poznámka: Polia v jazyku C++/CLI sú, ako by sme zrejme očakávali, indexované od nuly. Na prístup k i-tému prvku poľa použijeme výraz **p[i]**, kde **p** je identifikátor odkazovej premennej (ktorá uchováva odkaz na cieľové pole) a **i** je diskrétna celočíselná konštanta, ktorá reprezentuje pozíciu požadovaného prvku v poli (pričom samozrejme platí, že $0 \leq i < n$).

Upozornenie: Považujeme za dôležité poznamenať, že definícia odkazovej premennej neimplikuje inštanciáciu jednorozmerného poľa. Teda príkaz

```
array<int> ^p;
```

síce zakladá v zásobníku primárneho programového vlákna odkazovú premennú so symbolickým pomenovaním **p**, no neiniciuje alokáciu poľa. Po spracovaní tohto príkazu je hodnota premennej **p** nedefinovaná, a preto musí ešte pred použitím tejto premennej vo výraze alebo príkaze dôjsť k jej inicializácii, a to napr. nasledujúcim spôsobom:

```
p = gcnew array<int>(4);
```

Prekladač jazyka C++/CLI vyžaduje, aby bola medzi dátovými typmi odkazovej premennej a poľa úplná zhoda. Pri odloženej inicializácii odkazovej premennej sa niekedy môže stať, že sa programátor pomýli a pri inštanciácii poľa uvedie nekompatibilný dátový typ. Vďaka promptnému zásahu prekladača sa darí takéto druhy chýb okamžite zachytiť a eliminovať tak ich výskyt v zdrojovom kóde.

Podľa smerníc, ktoré nám velia písať priezračne čistý zdrojový kód, sa pri vytváraní polí snažíme spravidla vždy aplikovať operátor **gcnew**, aby vývojári získali vizuálnu spätnú väzbu, ktorá podáva informáciu o generovaní nového poľa. No pravda je, že pole môžeme vytvoriť aj bez explicitného použitia tohto inštanciačného operátora.

To sa deje v nasledujúcom príkaze:

```
array<double> ^p = {1.13, 2.78, 3.57};
```

Predstavený príkaz spolupracuje s inicializačným zoznamom, ktorý je tvorený konečnou a neprázdnou množinou diskrétnych inicializačných hodnôt (tzv. inicializátorov). Prekladač postupuje takto: najskôr skontroluje dátové typy inicializátorov. Ak sú typy kompatibilné, určí absolútnu početnosť inicializátorov, čím v skutočnosti stanoví rozsah novo vytváraného poľa. V ďalšej etape prekladač použije operátor **gcnew**, pomocou ktorého pole v riadenej halde alokuje. Do prvkov poľa potom nakopíruje inicializačné hodnoty a následne poskytne odkaz na (v tomto okamihu už) úspešne alokované a inicializované pole. Vrátený odkaz je nakoniec priradený do príslušnej odkazovej premennej.

V každom prípade, vždy, keď dochádza k vytvoreniu riadeného poľa, emituje prekladač jazyka C++/CLI inštrukciu **newarr [mscorlib]T** jazyka MSIL, kde **T** je dátový typ riadeného poľa. Čiže bez ohľadu na to, či je operátor **gcnew** v zdrojovom kóde explicitne použitý, alebo nie, prekladač s ním stále spolupracuje.

Tip: Objektová povaha polí je prínosná, pretože nám umožňuje používať metódy a vlastností polí. Nasledujúci fragment zdrojového kódu jazyka C++/CLI demonštruje využitie skalárnej inštančnej vlastnosti **Length**, ktorá udáva počet prvkov poľa:

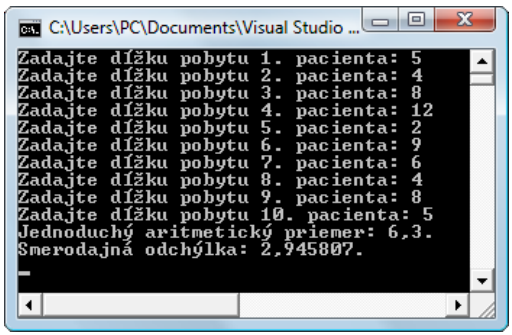
```
int main(array<System::String ^> ^args)
{
    array<double> ^p = {1.13, 2.78, 3.57};
    for(int i = 0; i < p->Length; i++)
    {
        Console::WriteLine(p[i].ToString());
    }
    Console::Read();
    return 0;
}
```

Hodnoty prvkov vektorového poľa sú automaticky získané a odoslané do výstupného dátového prúdu.

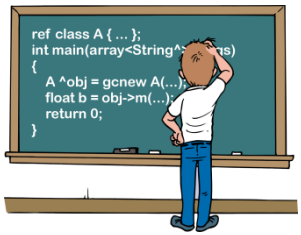
V záujme optimalizácie vyššie uvedeného fragmentu zdrojového kódu môžeme zapojiť do hry cyklus **for each**:

```
int main(array<System::String ^> ^args)
{
    array<double> ^p = {1.13, 2.78, 3.57};
    for each(double i in p)
    {
        Console::WriteLine(i.ToString());
    }
    Console::Read();
    return 0;
}
```

Keď riadený program spustíme a zadáme testovacie vstupné dáta, získame na výstupe hodnoty dvoch štatistických ukazovateľov: jednoduchého aritmetického priemeru a smerodajnej odchýlky (obr. 8).



Obr. 8: Výstup programu na výpočet smerodajnej odchýlky výberového štatistického súboru dát

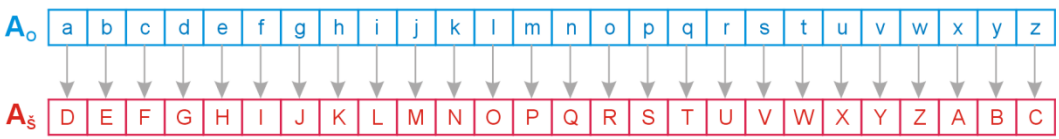


6 Praktický príklad č. 2: Program na šifrovanie a dešifrovanie textových dát pomocou Cézarovej šifry

Vedomostná náročnosť:

Časová náročnosť: **20** minút

Cézarova šifra je monoalfabetická substitučná šifra. Šifrovanie správy otvoreného textu sa uskutočňuje tak, že každý znak otvorenej abecedy (A_O) je nahradený práve jedným znakom šifrovej abecedy ($A_Š$). Medzi znakom otvorenej abecedy a znakom šifrovej abecedy existuje konštantný posun n znakov smerom doprava. Pri Cézarovej šifre platí, že $n = 3$. Otvorenou abecedou je štandardná medzinárodná abeceda, ktorá má 26 znakov. Šifrovú abecedu získame, keď každý znak otvorenej abecedy substituujeme znakom, ktorý sa nachádza o 3 pozície ďalej smerom doprava. Posun je pritom cyklický, čo znamená, že po dosiahnutí konca abecedy je realizovaný presun zase na jej začiatok (obr. 9).



Obr. 9: Otvorená (A_O) a šifrová ($A_Š$) abeceda pri použití Cézarovej šifry

Množinou vstupných dát pre šifrovací algoritmus bude správa otvoreného textu (Z_{OT}):

$$Z_{OT} = \{a_1, a_2, \dots a_n\}$$
$$a_i \text{ je i-tý znak } A_O, \forall i \in < 1, n >$$

Množinou výstupných dát, ktorú bude šifrovací algoritmus vytvárať, je správa šifrového textu ($Z_{ŠT}$):

$$Z_{ŠT} = \{s_1, s_2, \dots s_n\}$$
$$s_i \text{ je i-tý znak } A_Š, \forall i \in < 1, n >$$

Šifrovací algoritmus je predstavovaný šifrovacou transformačnou funkciou Ef:

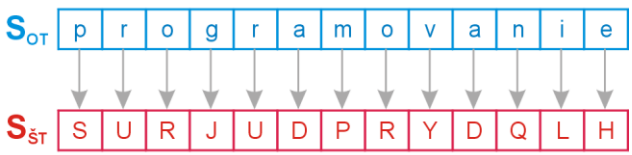
$$Ef(i)\colon Z_{OT} \rightarrow Z_{ŠT}$$

$$Ef(i): a_i + k$$
$$k = 3, \forall i \in < 1, n >$$

Dešifrovací algoritmus vyjadruje dešifrovacia transformačná funkcia Df:

$$Df(i): Z_{ŠT} \rightarrow Z_{OT}$$
$$Df(i): s_i - k$$
$$k = 3, \forall i \in < 1, n >$$

Praktický príklad: Povedzme, že budeme chcieť pomocou Cézarovej šifry zašifrovať nasledujúcu správu otvoreného textu: „programovanie“[2]. Budeme postupovať tak, že každý znak otvorenej abecedy (ktorá formuje otvorený text) nahradíme znakom šifrovej abecedy s trojitým posunom (obr. 10).



Obr. 10: Správu otvoreného textu (S_{OT}) sme previedli na správu šifrového textu ($S_{ŠT}$) pomocou Cézarovej šifry

Výsledkom šifrovacieho procesu je šifrový text „SURJUDPRYDQLH“.

Praktická algoritmizácia programu:

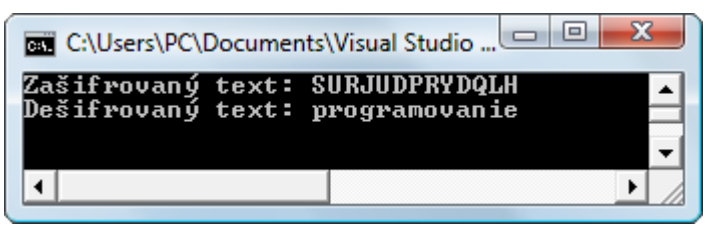
```
#include "stdafx.h"

using namespace System;

// Deklarácia triedy, ktorá bude zapuzdrovať funkcionalitu
// na šifrovanie a dešifrovanie správ.
ref class CezarovaSifra
{
private:
    // Uchovanie otvorenej abecedy vo vektorovom poli.
    static array<wchar_t> ^otvorenaAbeceda = {
```

[2] Podľa konvencie platí, že otvorený text sa zapisuje malými písmenami otvorenej abecedy, zatiaľ čo šifrový text sa zapisuje veľkými písmenami šifrovej abecedy.

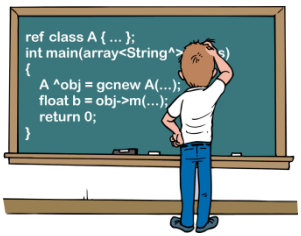
```
        'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h',
        'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p',
        'q', 'r', 's', 't', 'u', 'v', 'w', 'x',
        'y', 'z', ' '
    };

    // Uchovanie šifrovej abecedy vo vektorovom poli.
    static array<wchar_t> ^sifrovaAbeceda = {
        'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K',
        'L', 'M', 'N', 'O', 'P', 'Q', 'R', 'S',
        'T', 'U', 'V', 'W', 'X', 'Y', 'Z', 'A',
        'B', 'C', ' '
    };

    array<wchar_t> ^sifrovyText, ^otvorenyText;
public:
    // Definícia šifrovacej metódy.
    void Zasifrovat(String ^otvorenyText)
    {
        otvorenyText = otvorenyText->ToLower();
        sifrovyText = gcnew array<wchar_t>(otvorenyText->Length + 1);
        for(int i = 0; i < otvorenyText->Length; i++)
        {
            wchar_t oPismeno = otvorenyText[i];
            wchar_t sPismeno;
            for(int j = 0; j < otvorenaAbeceda->Length; j++)
            {
                if(otvorenaAbeceda[j] == oPismeno)
                sPismeno = sifrovaAbeceda[j];
            }
            sifrovyText[i] = sPismeno;
        }
        sifrovyText[sifrovyText->Length - 1] = '\0';
        Console::Write(sifrovyText);
    }
    // Definícia dešifrovacej metódy.
    void Desifrovat(String ^sifrovyText)
    {
        sifrovyText = sifrovyText->ToUpper();
        otvorenyText = gcnew array<wchar_t>(sifrovyText->Length + 1);
        for(int i = 0; i < sifrovyText->Length; i++)
        {
            wchar_t sPismeno = sifrovyText[i];
            wchar_t oPismeno;
            for(int j = 0; j < sifrovaAbeceda->Length; j++)
            {
                if(sifrovaAbeceda[j] == sPismeno)
```

```
                    oPismeno = otvorenaAbeceda[j];
                }
                otvorenyText[i] = oPismeno;
            }
            otvorenyText[otvorenyText->Length - 1] = '\0';
            Console::Write(otvorenyText);
        }
};

int main(array<System::String ^> ^args)
{
    // Inštanciácia deklarovanej triedy.
    CezarovaSifra ^sifra = gcnew CezarovaSifra();
    Console::Write("Zašifrovaný text: ");
    // Šifrovanie otvoreného textu pomocou Cézarovej šifry.
    sifra->Zasifrovat("programovanie");
    Console::WriteLine();
    Console::Write("Dešifrovaný text: ");
    // Dešifrovanie šifrového textu pomocou Cézarovej šifry.
    sifra->Desifrovat("SURJUDPRYDQLH");
    Console::Read();
    return 0;
}
```

Komentár k zdrojovému kódu: Programovú funkcionalitu na šifrovanie a dešifrovanie správ zapuzdríme do triedy **CezarovaSifra**. V súkromnej sekcii triedy definujeme dve statické vektorové polia dátového typu **wchar_t**, ktoré obsahujú lineárne postupnosti znakov otvorenej a šifrovej abecedy. Vo verejnej sekcii triedy sú situované definície dvoch inštančných parametrických metód: **Zasifrovat** a **Desifrovat**. Obe metódy vykonávajú transformácie so správami, pričom využívame veľmi jednoduchú a priamočiaru implementáciu šifrovacieho a dešifrovacieho procesu. V tele hlavnej metódy **main** inštanciujeme triedu **CezarovaSifra** a voláme metódy na šifrovanie a dešifrovanie testovacích správ. Výsledky, ku ktorým sa tieto metódy dopracujú, zobrazujeme na výstupe (obr. 11).



Obr. 11: Výstup programu, ktorý uskutočňuje šifrovanie a dešifrovanie textových dát pomocou Cézarovej šifry



7 Praktický príklad č. 3: Program na určenie rovnovážneho stavu spotrebiteľa

Vedomostná náročnosť:

Časová náročnosť: **25** minút

Jednou z najatraktívnejších oblastí modernej všeobecnej ekonomickej teórie je teória užitočnosti, ktorá sa venuje skúmaniu správania spotrebiteľa pri voľbe optimálnej spotrebnej stratégie. Spotrebiteľ sa správa pri spotrebúvaní statkov a služieb rozumne, a to v tom zmysle, že sa snaží maximalizovať užitočnosť, ktorú mu spotreba vybraných statkov a služieb prináša. Ekonomické vedy definujú užitočnosť ako mieru uspokojenia potrieb spotrebiteľa, ktorá je vyvolaná spotrebou určitého množstva statkov. Základy teórie užitočnosti boli položené ešte v 19. storočí, no z dnešného pohľadu dokážeme identifikovať dva hlavné prístupy, ktoré sa skúmaniu teórie užitočnosti venujú. Ide o kardinálny a ordinálny prístup. V ďalšom výklade budeme vychádzať z kardinálneho prístupu, podľa ktorého spotrebiteľ dokáže exaktne a absolútne kvantifikovať užitočnosť, ktorú mu poskytuje spotreba vybraných statkov. Na druhej strane, ordinálny prístup vychádza z predpokladu, že spotrebiteľ nie je schopný presnej kvantifikácie užitočnosti, ktorú mu prináša konzumácia rôznych statkov. Hoci zbavený schopnosti exaktnej kvantifikácie užitočnosti, spotrebiteľ dokáže špecifikovať, spotreba ktorých statkov (resp. ich kombinácií) mu prinesie väčšie uspokojenie (a je teda pre neho „viac užitočná“).

Podľa kardinálneho prístupu platí, že spotrebiteľ vie merať užitočnosť, ktorá je generovaná spotrebou určitého množstva statkov. Užitočnosť, ktorú pre spotrebiteľa produkuje konzumácia istého množstva statkov, sa vyjadruje v špeciálnych jednotkách, ktoré niektorí ekonómovia označujú ako „utily“. Napr. spotrebiteľ môže konzumácii jedného hamburgera priradiť užitočnosť 6 jednotiek (utilov), zatiaľ čo spotrebe jednej tabuľky mliečnej čokolády smie priradiť užitočnosť 8 jednotiek (utilov).

Celková užitočnosť (angl. *Total Utility*, *TU*) predstavuje celkové uspokojenie potrieb spotrebiteľa pri spotrebe n jednotiek statku. Celková užitočnosť má rastúcu tendenciu, čo znamená, že jej hodnota sa so vzrastajúcim množstvom spotrebovaného statku zvyšuje. Hraničná užitočnosť (angl. *Marginal Utility*, *MU*) determinuje, o koľko sa zvýši celková užitočnosť (generovaná spotrebou statku) pre spotrebiteľa, ak ten spotrebuje ďalšiu jednotku tohto statku. Hraničná užitočnosť je pre ekonómov významná, pretože im umožňuje zistiť, ako sa spotreba ďalšej jednotky statku odrazí na celkovom uspokojení jeho potrieb. Zatiaľ čo celková užitočnosť má rastovú tendenciu, hraničná užitočnosť s každou

spotrebovanou jednotkou statku klesá. V ekonómii je tento jav charakterizovaný zákonom klesajúcej hraničnej užitočnosti. Samozrejme, celková užitočnosť nerastie do nekonečna: jej kumulatívny priebeh sa končí v okamihu, keď je dosiahnutý bod nasýtenia. Bod nasýtenia znamená maximálnu celkovú užitočnosť, čiže maximálne uspokojenie potrieb spotrebiteľa, ktoré je vyvolané spotrebou n jednotiek statku. Len čo spotrebiteľ dosiahne bod nasýtenia, tak s každou ďalšou spotrebovanou jednotkou statku bude jeho celková užitočnosť klesať (a hraničná užitočnosť sa stane zápornou).

Praktický príklad: Uvažujme, že spotrebiteľ chce zahnať hlad, pričom na výber má dva statky: hamburger a čokoládovú tyčinku. V tab. 3 uvádzame, akú celkovú a hraničnú užitočnosť generuje pre spotrebiteľa konzumácia n hamburgerov a čokoládových tyčiniek, pričom $0 < n \leq 5$.

Tab. 3: Užitočnosť pri spotrebe hamburgerov a čokoládových tyčiniek

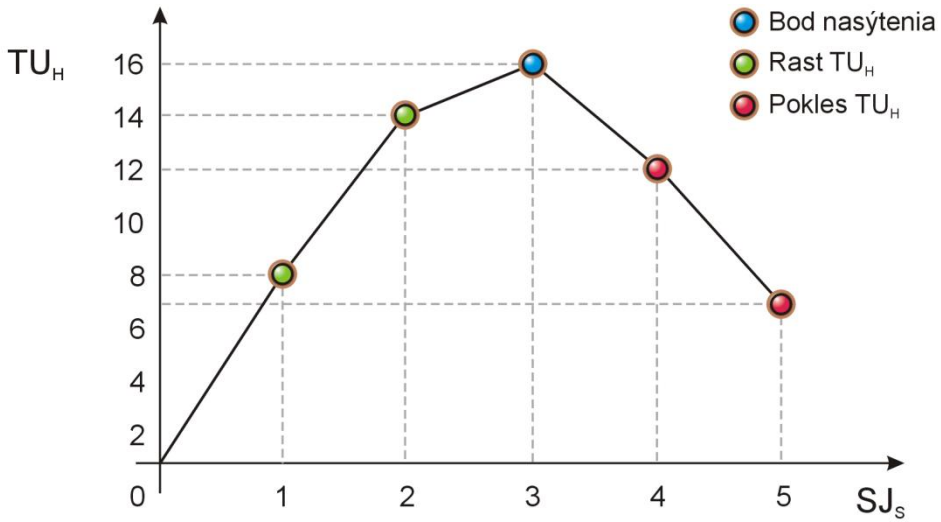
SJ_S	TU_H	MU_H	$TU_{ČT}$	$MU_{ČT}$
1	8	8	7	7
2	14	6	13	6
3	16	2	15	2
4	12	-4	14	-1
5	7	-5	9	-5

Legenda k tab. 3:

- **SJ_S** sú spotrebované jednotky hamburgerov a čokoládových tyčiniek.
- **TU_H** je celková užitočnosť plynúca z konzumácie hamburgerov.
- **MU_H** je hraničná užitočnosť plynúca z konzumácie ďalšieho hamburgera.
- **$TU_{ČT}$** je celková užitočnosť plynúca z konzumácie čokoládových tyčiniek.
- **$MU_{ČT}$** je hraničná užitočnosť plynúca z konzumácie ďalšej čokoládovej tyčinky.

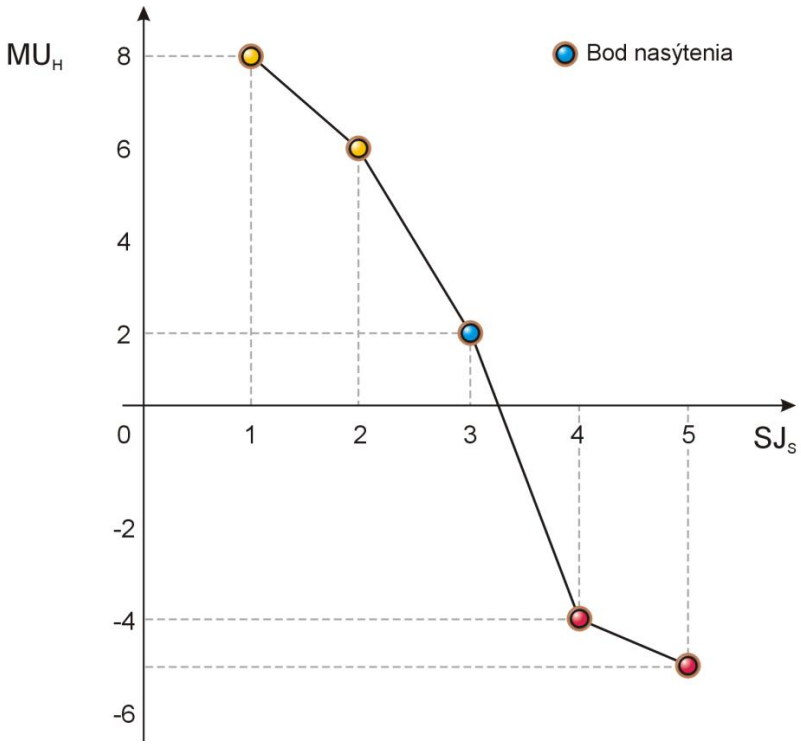
Komentár k tab. 3: Keď spotrebiteľ zje prvý hamburger, tak získa najväčší pocit uspokojenia. Ten je vyjadrený 8 utilmi. Keďže ide o prvý hamburger, tak celková užitočnosť je rovnaká ako hraničná užitočnosť (8). Jeden hamburger však nemusí utíšiť hlad úplne, a preto si spotrebiteľ objedná i druhú porciu. Tá ho zasýti viac: celková užitočnosť zo zjedených hamburgerov sa zvýši z 8 na 14 utilov. Na druhej strane, druhý hamburger nemá

schopnosť uspokojiť spotrebiteľa v takej miere ako prvý. Tento fakt dokumentuje pokles hraničnej užitočnosti, ktorá sa z 8 utilov znižuje na 6 utilov. Po konzumácii tretieho hamburgera vidíme, že spotrebiteľ dosiahol bod nasýtenia. Bod nasýtenia opisuje maximálna hodnota celkovej užitočnosti, ktorá je v našom prípade 16 utilov. Vzhľadom na platiaci zákon klesajúcej hraničnej užitočnosti nie je prekvapujúce, že dosiahnutie bodu nasýtenia prináša len 2 dodatočné jednotky užitočnosti. Ako sme už uviedli, po dosiahnutí bodu nasýtenia celková užitočnosť klesá a hraničná užitočnosť sa zakrátko dostane do pásma záporných hodnôt. Tieto fakty mapujú obr. 12 – 13.



Obr. 12: Vizualizácia celkovej užitočnosti, ktorú generuje spotreba hamburgerov

Teória užitočnosti vraví, že keď spotrebiteľ skonzumuje tri hamburgery, tak dosiahne najväčšiu celkovú užitočnosť. Napriek tomu sa môže stať, že spotrebiteľ sa rozhodne aj pre zjedenie ďalšieho, v poradí už štvrtého hamburgera. No vzhľadom na to, že TU_H poklesne o 4 utily, je isté, že spotrebiteľ zažije skôr nepríjemný pocit preplneného žalúdka, ako zvýšenie svojej subjektívnej spokojnosti. Ako vidíme, s poklesom TU_H sa MU_H dostáva do pásma záporných hodnôt. Ak by sa spotrebiteľ rozhodol jesť iba hamburgery, tak optimum by dosiahol pri 3 spotrebovaných jednotkách tohto statku.



Obr. 13: Vizualizácia hraničnej užitočnosti, ktorú generuje spotreba hamburgerov

Aká je však situácia pri čokoládových tyčinkách? Po analýze zisťujeme, že celková užitočnosť, plynúca z konzumácie čokoládových tyčiniek, dosahuje maximum 15 utilov pri spotrebe 3 kusov tejto dobroty. Zaujímavé je, že ak spotrebiteľ zje 4 tyčinky, tak celková užitočnosť poklesne len o 1 util (a hraničná užitočnosť bude mať hodnotu -1). Ak by spotrebiteľ chcel zahnať hlad len konzumáciou čokoládových tyčiniek, tak je zrejmé, že ako optimum sa pre spotrebiteľa javia 3 kusy.

Oveľa zaujímavejšie však bude, keď dokážeme nájsť odpoveď na nasledujúcu otázku: Aká bude optimálna stratégia spotrebiteľa, ktorý sa rozhodne konzumovať súčasne hamburgery a čokoládové tyčinky? Povedané inak, aká kombinácia spotrebovaných jednotiek hamburgerov a čokoládových tyčiniek bude pre spotrebiteľa najlepšia? Nuž, najlepšia bude určite taká kombinácia týchto statkov, ktorá spotrebiteľovi prinesie maximálne uspokojenie. V ekonómii hovoríme, že rovnovážny stav spotrebiteľ dosiahne vtedy, ak maximalizuje hraničné užitočnosti na jednotkové ceny statkov. Aby sme mohli pokračovať v našich

úvahách ďalej, budeme predpokladať, že jeden hamburger stojí 2 € (P_H = 2 €), zatiaľ čo na nákup jednej čokoládovej tyčinky je nutné vynaložiť 50 centov ($P_{ČT}$ = 0,5 €). Tab. 4 monitoruje pomer MU/P u oboch statkov.

Tab. 4: Analýza pomerov MU/P pri spotrebovaných statkoch (P_H = 2 €, $P_{ČT}$ = 0,5 €)

SJ_S	TU_H	MU_H	MU_H/P_H	$TU_{ČT}$	$MU_{ČT}$	$MU_{ČT}/P_{ČT}$
1	8	8	4	7	7	14
2	14	6	3	13	6	12
3	16	2	1	15	2	4
4	12	-4	-2	14	-1	-2
5	7	-5	-2,5	9	-5	-10

Spotrebiteľ sa snaží svoju užitočnosť maximalizovať, a preto volí také množstvá spotrebovaných statkov, ktoré mu prinesú maximálne uspokojenie. Reálne možnosti spotrebiteľa sú však obmedzené jeho disponibilným príjmom (resp. časťou disponibilného príjmu), ktorý je spotrebiteľ ochotný vynaložiť na nákup požadovaného množstva statkov.

Rovnovážny stav spotrebiteľa nastáva vtedy, keď sa pomer hraničných užitočností spotrebovaných hamburgerov a čokoládových tyčiniek rovná pomeru ich jednotkových cien. Matematicky zapísané:

$$\frac{MU_H}{MU_{ČT}} = \frac{P_H}{P_{ČT}} \Rightarrow \frac{MU_H}{P_H} = \frac{MU_{ČT}}{P_{ČT}}$$

kde:

- MU_H je hraničná užitočnosť, ktorá sa viaže so spotrebou ďalšieho hamburgera.
- $MU_{ČT}$ je hraničná užitočnosť, ktorá sa viaže so spotrebou ďalšej čokoládovej tyčinky.
- P_H je cena jedného hamburgera.
- $P_{ČT}$ je cena jednej čokoládovej tyčinky.

Z tab. 4 zisťujeme, že spotrebiteľ optimalizuje svoju voľbu a dosiahne rovnovážny stav vtedy, keď skonzumuje jeden hamburger a tri čokoládové tyčinky.

7.1 Matematicko-ekonomický algoritmus aplikovaný pri analýze rovnovážneho stavu spotrebiteľa

V záujme vysvetlenia všeobecného matematicko-ekonomického algoritmu zavedieme nasledujúci abstraktný ekonomický model:

1. Budeme pracovať s dvomi statkami, pričom prvý označíme identifikátorom **A** a druhý identifikátorom **B**.

2. Cenu jednej spotrebiteľskej jednotky statku **A** označíme identifikátorom $\mathbf{P_A}$. Analogicky, cene jednej spotrebiteľskej jednotky statku **B** prisúdime identifikátor $\mathbf{P_B}$.

3. Časť disponibilného príjmu spotrebiteľa, ktorá bude alokovaná na nákup statkov **A** a **B**, označíme identifikátorom **I**. Príjem **I** bude slúžiť len na nákup statkov **A** a **B**. Pritom však musíme pamätať na to, že spotrebiteľ sa bude usilovať kúpiť takú kombináciu statkov **A** a **B**, ktorá zabezpečí maximálne uspokojenie jeho potrieb. Vzhľadom na skutočnosť, že spotrebiteľ je obmedzený výškou svojho príjmu **I**, je zrejmé, že nebude môcť nakúpiť viac jednotiek statkov **A** a **B**, než aké mu dovoľuje nasledujúca rovnica:

$$A \times P_A + B \times P_B = I$$

   kde:
   - A je spotrebované množstvo statku **A**.
   - B je spotrebované množstvo statku **B**.

4. Naším cieľom je zistiť, kedy spotrebiteľ dosiahne rovnovážny stav. Rovnovážny stav charakterizuje optimum spotrebiteľa, teda stav, v ktorom spotrebiteľ získa maximálny úžitok zo spotrebovaných statkov. Tento stav nastáva vtedy, keď sa pomer hraničných užitočností spotrebovaných statkov rovná pomeru ich jednotkových cien.

Matematicky zapísané:

$$\frac{MU_A}{MU_B} = \frac{P_A}{P_B} \Rightarrow \frac{MU_A}{P_A} = \frac{MU_B}{P_B}$$

kde:

- MU_A je hraničná užitočnosť, ktorá sa viaže so spotrebou dodatočnej jednotky statku **A**.
- MU_B je hraničná užitočnosť, ktorá sa viaže so spotrebou dodatočnej jednotky statku **B**.

5. Funkcia užitočnosti statkov **A** a **B** nech je takáto: $U = f(A, B) = A \times B$.
6. Hraničné užitočnosti statkov **A** a **B** určíme pomocou ich parciálnych derivácií:

$$MU_A = \frac{\partial U}{\partial A} = \frac{\partial f(A,B)}{\partial A}$$

$$MU_B = \frac{\partial U}{\partial B} = \frac{\partial f(A,B)}{\partial B}$$

Praktický príklad: Spotrebiteľ sa chystá vynaložiť na nákup statkov **A** a **B** dovedna 100 €. Funkcia užitočnosti oboch statkov je $U = f(A, B) = A \times B$. Cena jednej spotrebiteľskej jednotky statku **A** je 4 € ($P_A = 4$ €). Cena jednej spotrebiteľskej jednotky statku **B** je 10 € ($P_B = 10$ €). Koľko jednotiek statkov **A** a **B** spotrebiteľ nakúpi, aby maximalizoval svoju užitočnosť?

Postup riešenia úlohy je nasledujúci:

1. Stanovíme rovnicu pre rozpočtové obmedzenie: $4 \times A + 10 \times B = 100$.
2. Chceme maximalizovať funkciu užitočnosti: $U = A \times B \rightarrow \max$.
3. Vypočítame hraničné užitočnosti statkov **A** a **B**:

$$MU_A = \frac{\partial U}{\partial A} = \frac{\partial A \times B}{A} = B$$
$$MU_B = \frac{\partial U}{\partial B} = \frac{\partial A \times B}{B} = A$$

4. Rovnovážny stav spotrebiteľa nastane vtedy, keď:

$$\frac{MU_A}{P_A} = \frac{MU_B}{P_B}$$

$$\frac{B}{P_A} = \frac{A}{P_B} = \frac{B}{4} = \frac{A}{10} \Rightarrow A = \frac{10 \times B}{4}$$

5. Získame tieto dve rovnice (r_1, r_2):

$r_1: 4 \times A + 10 \times B = 100$
$r_2: A = \frac{10 \times B}{4}$

6. Dosadením r_2 do r_1 dostávame:

$$4 \times \frac{10 \times B}{4} + 10 \times B = 100$$
$$10 \times B + 10 \times B = 100$$
$$20 \times B = 100$$
$$B = 5$$

7. Dosadením **B** do r_2 získame: $A = \frac{10 \times B}{4} = \frac{10 \times 5}{4} = \frac{50}{4} = 12{,}5$.

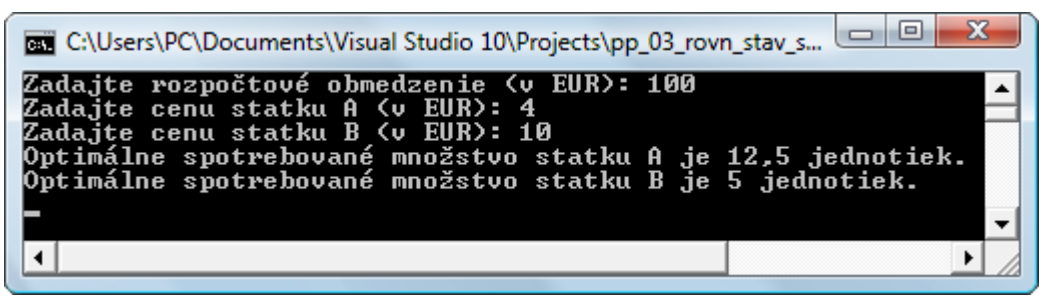
8. Spotrebiteľ dosiahne rovnovážny stav pri konzumácii 12,5 spotrebiteľských jednotiek statku **A** a 5 spotrebiteľských jednotiek statku **B**.

Praktická algoritmizácia programu:

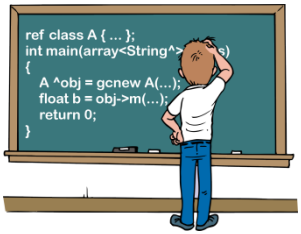
```
#include "stdafx.h"

using namespace System;

int main(array<System::String ^> ^args)
{
    // Definície premenných;
    float jednotkyStatkuA, jednotkyStatkuB;
    float rozpoctoveObmedzenie;
    float cenaStatkuA, cenaStatkuB;
```

```
    // Inicializácia premenných podľa používateľských vstupov.
    Console::Write("Zadajte rozpočtové obmedzenie (v EUR): ");
    rozpoctoveObmedzenie = Convert::ToSingle(Console::ReadLine());
    Console::Write("Zadajte cenu statku A (v EUR): ");
    cenaStatkuA = Convert::ToSingle(Console::ReadLine());
    Console::Write("Zadajte cenu statku B (v EUR): ");
    cenaStatkuB = Convert::ToSingle(Console::ReadLine());
    // Detekcia optimálneho množstva spotrebovaných jednotiek statkov A a B.
    jednotkyStatkuB = rozpoctoveObmedzenie / (2 * cenaStatkuB);
    jednotkyStatkuA = (cenaStatkuB * jednotkyStatkuB) / cenaStatkuA;
    // Zobrazenie vypočítaných údajov na výstupe.
    Console::WriteLine("Optimálne spotrebované množstvo statku A je " +
        "{0} jednotiek.", jednotkyStatkuA);
    Console::WriteLine("Optimálne spotrebované množstvo statku B je " +
        "{0} jednotiek.", jednotkyStatkuB);
    Console::Read();
    return 0;
}
```

Komentár k zdrojovému kódu: Po načítaní vstupných hodnôt od používateľa (predmetom načítania je rozpočtové obmedzenie a ceny statkov **A** a **B**) program zisťuje optimálnu kombináciu spotrebiteľských jednotiek statkov **A** a **B**, ktoré pri danom rozpočtovom obmedzení maximalizujú spotrebiteľovu spokojnosť. Pripomíname, že funkcia užitočnosti má takú podobu, ako sme uviedli vyššie, teda $U = A \times B$. Výstup programu je znázornený na obr. 14.



Obr. 14: Výstup programu na určenie rovnovážneho stavu spotrebiteľa



8 Praktický príklad č. 4: Program na riešenie sústavy 3 lineárnych rovníc s 3 neznámymi pomocou determinantov

Vedomostná náročnosť:

Časová náročnosť: **30** minút

V matematike, ale aj v iných vedách, sa veľmi často stretávame s problémom nájdenia množiny koreňov sústavy n lineárnych rovníc s n neznámymi. V tomto praktickom cvičení vás zoznámime s algoritmizáciou riešenia sústavy 3 lineárnych rovníc s 3 neznámymi pomocou determinantov a tzv. Cramerovho pravidla.

Akúkoľvek sústavu lineárnych rovníc vieme vyjadriť v maticovom tvare:

1. Majme sústavu 3 lineárnych rovníc s 3 neznámymi:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 &= b_1 \\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 &= b_2 \\ a_{31}x_1 + a_{32}x_2 + a_{33}x_3 &= b_3 \end{aligned}$$

2. Zavedieme nasledujúce 3 matice:

   a) matica koeficientov $\boldsymbol{A}$:

$$\boldsymbol{A} = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix}$$

   b) matica (vektor) neznámych $\boldsymbol{x}$:

$$\boldsymbol{x} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}$$

   c) matica (vektor) konštantných členov na pravej strane ($\boldsymbol{b}$):

$$\boldsymbol{b} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix}$$

3. Kompaktný maticový tvar sústavy 3 lineárnych rovníc s 3 neznámymi je takýto:

$$\boldsymbol{A} \times \boldsymbol{x} = \boldsymbol{b}.$$

Ak je matica $\boldsymbol{A}$ regulárna (teda ak platí, že $|\boldsymbol{A}| \neq 0$), tak sústava 3 lineárnych rovníc s 3 neznámymi má práve jedno riešenie, ktoré môžeme zapísať v tvare:

$$x_i = \frac{|\boldsymbol{A}_i|}{|\boldsymbol{A}|}, \forall i \in < 1,3 >$$

kde:

- $\boldsymbol{A}_i$ je matica, ktorá vznikne z matice koeficientov ($\boldsymbol{A}$). Pri konštrukcii matice $\boldsymbol{A}_i$ postupujeme tak, že i-tý stĺpec tejto matice substituujeme stĺpcom konštantných členov ($\boldsymbol{b}$).

Predstavený algoritmus riešenia sústavy lineárnych rovníc využíva determinanty a v matematike je známy ako Cramerovo pravidlo[3].

Praktický príklad: Vyriešme nasledujúcu sústavu 3 lineárnych rovníc s 3 neznámymi:

$$\begin{aligned} 3x_1 \quad\quad\quad + x_3 &= 6 \\ x_1 + 2x_2 + 3x_3 &= 14 \\ 4x_1 \quad\quad\quad + x_3 &= 7 \end{aligned}$$

1. Sústavu lineárnych rovníc prepíšeme do maticového tvaru:

$$\boldsymbol{A} \times \boldsymbol{x} = \boldsymbol{b}$$

$$\boldsymbol{A} = \begin{pmatrix} 3 & 0 & 1 \\ 1 & 2 & 3 \\ 4 & 0 & 1 \end{pmatrix}, \ \boldsymbol{x} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}, \boldsymbol{b} = \begin{pmatrix} 6 \\ 14 \\ 7 \end{pmatrix}$$

$$\begin{pmatrix} 3 & 0 & 1 \\ 1 & 2 & 3 \\ 4 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 6 \\ 14 \\ 7 \end{pmatrix}$$

[3] Pravidlo nesie meno Gabriela Cramera, významného švajčiarskeho matematika, ktorý žil a pôsobil v 18. storočí.

2. Vypočítame determinant matice koeficientov ($\boldsymbol{A}$):

$$|\boldsymbol{A}| = \begin{vmatrix} 3 & 0 & 1 \\ 1 & 2 & 3 \\ 4 & 0 & 1 \end{vmatrix} = -2$$

3. Vypočítame determinanty matíc $\boldsymbol{A_1}$, $\boldsymbol{A_2}$ a $\boldsymbol{A_3}$:

$$|\boldsymbol{A}_1| = \begin{vmatrix} 6 & 0 & 1 \\ 14 & 2 & 3 \\ 7 & 0 & 1 \end{vmatrix} = -2 \qquad |\boldsymbol{A}_2| = \begin{vmatrix} 3 & 6 & 1 \\ 1 & 14 & 3 \\ 4 & 7 & 1 \end{vmatrix} = -4$$

$$|\boldsymbol{A}_3| = \begin{vmatrix} 3 & 0 & 6 \\ 1 & 2 & 14 \\ 4 & 0 & 7 \end{vmatrix} = -6$$

4. Určíme množinu koreňov sústavy lineárnych rovníc $K = \{x_1,\ x_2,\ x_3\}$:

$$x_1 = \frac{|\boldsymbol{A}_1|}{|\boldsymbol{A}|} = \frac{-2}{-2} = 1$$

$$x_2 = \frac{|\boldsymbol{A}_2|}{|\boldsymbol{A}|} = \frac{-4}{-2} = 2$$

$$x_3 = \frac{|\boldsymbol{A}_3|}{|\boldsymbol{A}|} = \frac{-6}{-2} = 3$$

5. Riešením sústavy 3 lineárnych rovníc s 3 neznámymi je množina koreňov $K = \{1, 2, 3\}$.

Praktická algoritmizácia programu:

```
#include "stdafx.h"

using namespace System;

// Deklarácia triedy, ktorá zapuzdruje funkcionalitu na riešenie sústavy
// 3 lineárnych rovníc s 3 neznámymi.
ref class SustavaLinearnychRovnic3X3
{
```

```
private:
    // Definície súkromných dátových členov triedy.
    array<int, 2> ^maticaKoeficientov;
    array<int> ^maticaKonstantnychClenov;
    array<int, 2> ^matica_A1, ^matica_A2, ^matica_A3;
    int det_A, det_A1, det_A2, det_A3;
    int koren_x1, koren_x2, koren_x3;
public:
    // Definícia parametrického inštančného konštruktora.
    SustavaLinearnychRovnic3X3(array<int, 2> ^maticaKoeficientov,
        array<int> ^maticaKonstantnychClenov)
    {
        (*this).maticaKoeficientov = maticaKoeficientov;
        (*this).maticaKonstantnychClenov = maticaKonstantnychClenov;
        // Úprava 1. matice podľa Cramerovho pravidla.
        matica_A1 = gcnew array<int, 2>(3,3);
        for(int i = 0; i < 3; i++)
        {
            for(int j = 0; j < 3; j++)
            {
                matica_A1[i, j] = maticaKoeficientov[i, j];
            }
        }
        matica_A1[0,0] = maticaKonstantnychClenov[0];
        matica_A1[1,0] = maticaKonstantnychClenov[1];
        matica_A1[2,0] = maticaKonstantnychClenov[2];
        // Úprava 2. matice podľa Cramerovho pravidla.
        matica_A2 = gcnew array<int, 2>(3,3);
        for(int i = 0; i < 3; i++)
        {
            for(int j = 0; j < 3; j++)
            {
                matica_A2[i, j] = maticaKoeficientov[i, j];
            }
        }
        matica_A2[0,1] = maticaKonstantnychClenov[0];
        matica_A2[1,1] = maticaKonstantnychClenov[1];
        matica_A2[2,1] = maticaKonstantnychClenov[2];
        // Úprava 3. matice podľa Cramerovho pravidla.
        matica_A3 = gcnew array<int, 2>(3,3);
        for(int i = 0; i < 3; i++)
        {
            for(int j = 0; j < 3; j++)
            {
                matica_A3[i, j] = maticaKoeficientov[i, j];
            }
        }
```

```
    matica_A3[0,2] = maticaKonstantnychClenov[0];
    matica_A3[1,2] = maticaKonstantnychClenov[1];
    matica_A3[2,2] = maticaKonstantnychClenov[2];
}
// Definícia metódy na nájdenie koreňov sústavy 3 lineárnych rovníc
// s 3 neznámymi.
void Vyriesit()
{
    // Výpočet determinantu matice koeficientov.
    det_A = maticaKoeficientov[0,0] * maticaKoeficientov[1,1] *
        maticaKoeficientov[2,2] + maticaKoeficientov[0,1] *
        maticaKoeficientov[1,2] * maticaKoeficientov[2,0] +
        maticaKoeficientov[0,2] * maticaKoeficientov[1,0] *
        maticaKoeficientov[2,1] - maticaKoeficientov[0,2] *
        maticaKoeficientov[1,1] * maticaKoeficientov[2,0] -
        maticaKoeficientov[0,0] * maticaKoeficientov[1,2] *
        maticaKoeficientov[2,1] - maticaKoeficientov[0,1] *
        maticaKoeficientov[1,0] * maticaKoeficientov[2,2];
    // Výpočet determinantu 1. upravenej matice.
    det_A1 = matica_A1[0,0] * matica_A1[1,1] * matica_A1[2,2] +
        matica_A1[0,1] * matica_A1[1,2] * matica_A1[2,0] +
        matica_A1[0,2] * matica_A1[1,0] * matica_A1[2,1] -
        matica_A1[0,2] * matica_A1[1,1] * matica_A1[2,0] -
        matica_A1[0,0] * matica_A1[1,2] * matica_A1[2,1] -
        matica_A1[0,1] * matica_A1[1,0] * matica_A1[2,2];
    // Výpočet determinantu 2. upravenej matice.
    det_A2 = matica_A2[0,0] * matica_A2[1,1] * matica_A2[2,2] +
        matica_A2[0,1] * matica_A2[1,2] * matica_A2[2,0] +
        matica_A2[0,2] * matica_A2[1,0] * matica_A2[2,1] -
        matica_A2[0,2] * matica_A2[1,1] * matica_A2[2,0] -
        matica_A2[0,0] * matica_A2[1,2] * matica_A2[2,1] -
        matica_A2[0,1] * matica_A2[1,0] * matica_A2[2,2];
    // Výpočet determinantu 3. upravenej matice.
    det_A3 = matica_A3[0,0] * matica_A3[1,1] * matica_A3[2,2] +
        matica_A3[0,1] * matica_A3[1,2] * matica_A3[2,0] +
        matica_A3[0,2] * matica_A3[1,0] * matica_A3[2,1] -
        matica_A3[0,2] * matica_A3[1,1] * matica_A3[2,0] -
        matica_A3[0,0] * matica_A3[1,2] * matica_A3[2,1] -
        matica_A3[0,1] * matica_A3[1,0] * matica_A3[2,2];
    // Výpočet koreňov sústavy 3 lineárnych rovníc s 3 neznámymi.
    koren_x1 = det_A1 / det_A;
    koren_x2 = det_A2 / det_A;
    koren_x3 = det_A3 / det_A;
    // Vypočítané determinanty a korene sú zobrazené na výstupe.
    Console::WriteLine(" det_A = {0}.", det_A);
    Console::WriteLine("det_A1 = {0}.", det_A1);
    Console::WriteLine("det_A2 = {0}.", det_A2);
```

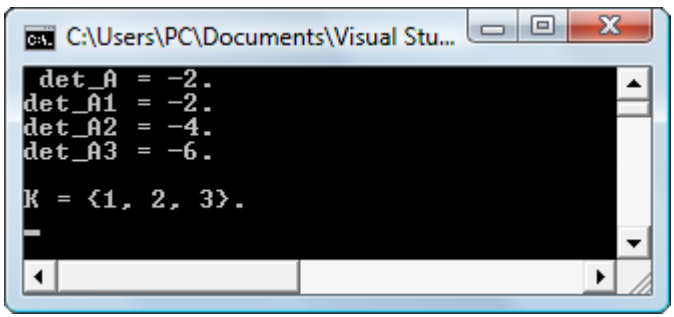
```
        Console::WriteLine("det_A3 = {0}.", det_A3);
        Console::WriteLine("\nK = {{{0}, {1}, {2}}}.", koren_x1,
            koren_x2, koren_x3);
    }
};
int main(array<System::String ^> ^args)
{
    // Definičná inicializácia 2D poľa,
    // v ktorom je uchovaná matica koeficientov.
    array<int, 2> ^mat_koef = gcnew array<int, 2>(3,3)
    {
        {3, 0, 1},
        {1, 2, 3},
        {4, 0, 1}
    };
    // Definičná inicializácia 1D poľa, v ktorom je uchovaná
    // matica konštantných členov pravej strany.
    array<int> ^mat_konst_clenov = gcnew array<int>(3) {6, 14, 7};
    SustavaLinearnychRovnic3X3 ^sustava;
    // Inštanciácia triedy.
    sustava = gcnew SustavaLinearnychRovnic3X3(mat_koef, mat_konst_clenov);
    // Inštancia triedy volá metódu, ktorá automatizuje proces nájdenia koreňov
    // sústavy 3 lineárnych rovníc s 3 neznámymi.
    sustava->Vyriesit();
    Console::Read();
    return 0;
}
```

Komentár k zdrojovému kódu: Všetku funkcionalitu, ktorá sa viaže s riešením sústavy 3 lineárnych rovníc s 3 neznámymi, sme umiestnili do triedy **SustavaLinearnychRovnic3X3**. V súkromnej sekcii triedy definujeme dátové členy, ktoré budeme potrebovať. Ide o maticu koeficientov, maticu konštantných členov pravej strany, trojicu matíc, ktoré budú upravované podľa Cramerovho pravidla, determinanty a korene sústavy lineárnych rovníc. Parametrický inštančný konštruktor triedy očakáva, že klientsky kód mu poskytne odkazy na maticu koeficientov a maticu konštantných členov pravej strany. Ako si môžeme všimnúť, matica koeficientov je syntakticky reprezentovaná dvojrozmerným (2D) celočíselným poľom. Keďže matica konštantných členov pravej strany je vektorom, stačí nám na jej uchovanie jednorozmerné (1D) pole. Okrem toho, že v tele konštruktora získame prístup k požadovaným vstupným maticiam, vytvárame ďalšie tri matice, ktoré inicializujeme podľa Cramerovho pravidla. To znamená, že i-tý stĺpec každej z týchto troch matíc je nahradený vektorom konštantných členov pravej strany sústavy.

Sústavu 3 lineárnych rovníc s 3 neznámymi rieši verejne prístupná, inštančná a bezparametrická metóda **Vyriesit**. Aby mohla metóda nájsť množinu koreňov, musí najskôr vypočítať determinant matice koeficientov a rovnako aj determinanty troch matíc modifikovaných podľa Cramerovho pravidla. V momente, keď sú hodnoty všetkých determinantov známe, metóda vypočíta korene sústavy a spoločne s determinantmi ich zobrazí na výstupe.

V tele hlavnej metódy **main** konštruujeme maticu koeficientov a maticu konštantných členov pravej strany sústavy 3 lineárnych rovníc s 3 neznámymi.

V ďalšej etape inštanciujeme triedu **SustavaLinearnychRovnic3X3**, pričom jej parametrickému konštruktoru odovzdávame korektné vstupné dáta (odkazy na polia, ktoré uchovávajú požadované matice). Napokon voláme metódu **Vyriesit**, čím nájdeme množinu koreňov sústavy 3 lineárnych rovníc s 3 neznámymi. Výstup programu ukazuje obr. 15.

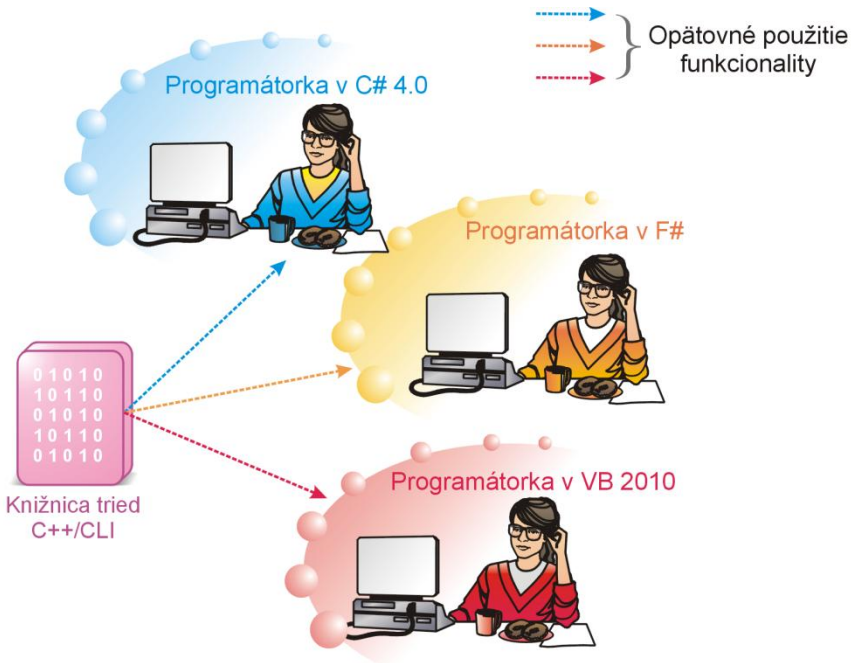


Obr. 15: Výstup programu, ktorý je schopný vyriešiť sústavu 3 lineárnych rovníc s 3 neznámymi

8.1 Demonštrácia interoperability jazykov C++/CLI a C# 4.0

Triedu **SustavaLinearnychRovnic3X3**, ktorú sme naprogramovali v jazyku C++/CLI, využijeme znova, pretože naším zámerom bude poukázať na efektívnu interoperabilitu programov, ktoré boli napísané v rôznych .NET-kompatibilných programovacích jazykoch. Pre naše potreby sme zvolili jazyky C++/CLI a C# 4.0, ale je samozrejme možné zvoliť akúkoľvek inú vhodnú kombináciu (napr. Visual Basic 2010 ↔ C# 4.0, C++/CLI ↔ Visual Basic 2010, či C++/CLI ↔ F#). Ako uvidíme, vzájomná spolupráca medzi jazykmi C++/CLI a C# 4.0 je veľmi produktívna, lebo nám umožňuje opätovne a predovšetkým okamžite

využiť zdrojový kód, ktorý bol napísaný v inom .NET-kompatibilnom programovacom jazyku (obr. 16).



Obr. 16: Maximalizácia pracovnej produktivity pomocou interoperability .NET-kompatibilných jazykov

Proces interoperability medzi jazykmi C++/CLI a C# 4.0 bude prebiehať v nasledujúcich etapách:

1. Triedu **SustavaLinearnychRovnic3X3** vložíme do knižnice tried. Knižnica tried je novým projektom jazyka C++/CLI, ktorý je určený na bezpečné zdieľanie funkcionality. Do knižníc tried sa vkladajú triedy, zapuzdrujúce istú vopred naprogramovanú funkcionalitu, ktorá bude opätovne využívaná klientmi. Z pohľadu cieľového vývojára, ktorý bude s knižnicou tried pracovať, je citeľnou konkurenčnou výhodou hlavne vyššia úroveň abstrakcie, ktorú táto knižnica zavádza. To je prospešné, pretože používateľ knižnice nemusí poznať presné technické detaily, ktoré determinujú bázovú funkcionalitu samotnej knižnice. Stačí, keď sa vývojár

naučí používať rozhranie, ktoré knižnica ponúka (rozhraním máme na mysli najmä triedy a ich metódy, ktoré sa v knižnici tried nachádzajú).

Keď prekladač jazyka C++/CLI zostaví knižnicu tried, vygeneruje súbor s príponou .dll. V tomto súbore bude uložený preložený MSIL kód knižnice tried. Súbor s knižnicou tried môže byť od tejto chvíle importovaný do projektu akéhokoľvek .NET-kompatibilného programovacieho jazyka a explicitne použitý.

2. Založíme nový projekt štandardnej konzolovej aplikácie jazyka C# 4.0.

3. Do projektu pridáme odkaz na knižnicu tried, ktorú sme vytvorili v jazyku C++/CLI.

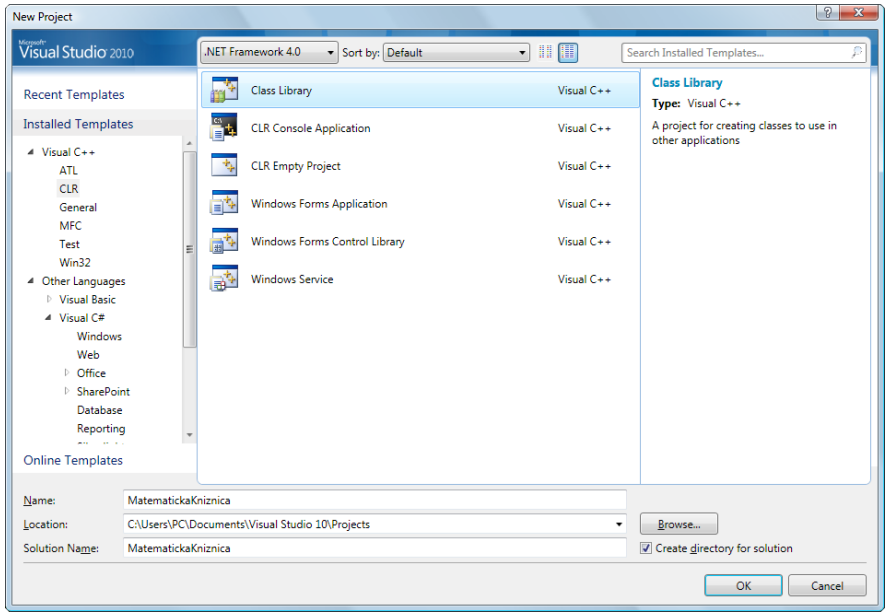
4. V jazyku C# 4.0 vytvoríme inštanciu triedy **SustavaLinearnychRovnic3X3** a použijeme ju na vyriešenie sústavy 3 lineárnych rovníc s 3 neznámymi.

8.1.1 Vytvorenie knižnice tried v jazyku C++/CLI

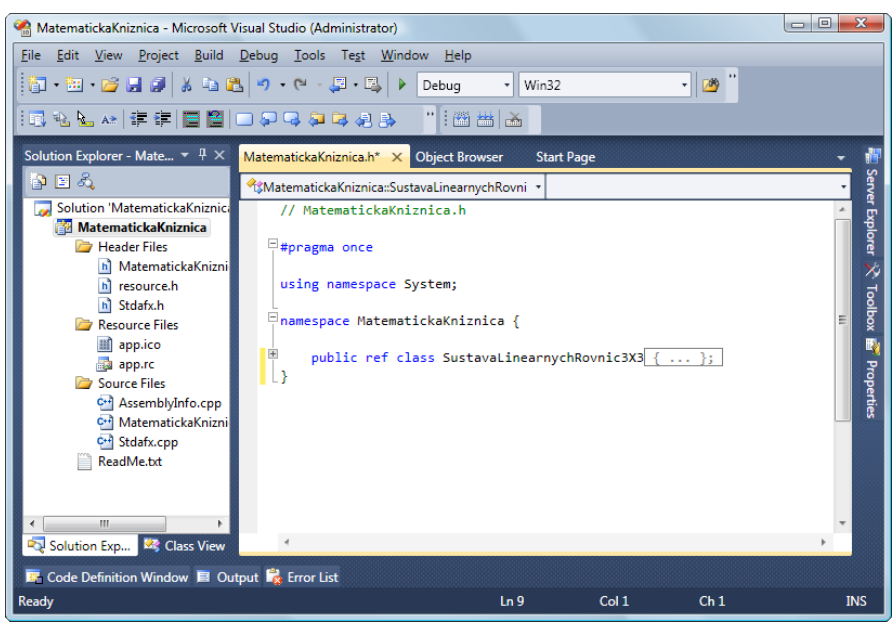
1. Na úvodnej stránke **Start Page** vyberieme položku **Projects** a klikneme na položku **New Project**. V dialógovom okne **New Project** zvolíme projektovú šablónu knižnice tried **Class Library** (**Visual C++ → CLR → Class Library**). Do textového poľa **Name** zapíšeme názov knižnice tried (v našom prípade MatematickaKniznica). V tomto okamihu by dialógové okno **New Project** malo vyzerať ako na obr. 17.

   Po kliknutí na tlačidlo **OK** sa spustí sprievodca, ktorý zhotoví nový projekt knižnice tried.

2. Do implicitne vytvoreného menného priestoru **MatematickaKniznica** vložíme deklaráciu verejnej triedy **SustavaLinearnychRovnic3X3**. Po tomto úkone bude mať zdrojový súbor (s identifikátorom MatematickaKniznica a príponou .h) podobu, akú zachytáva obr. 18.

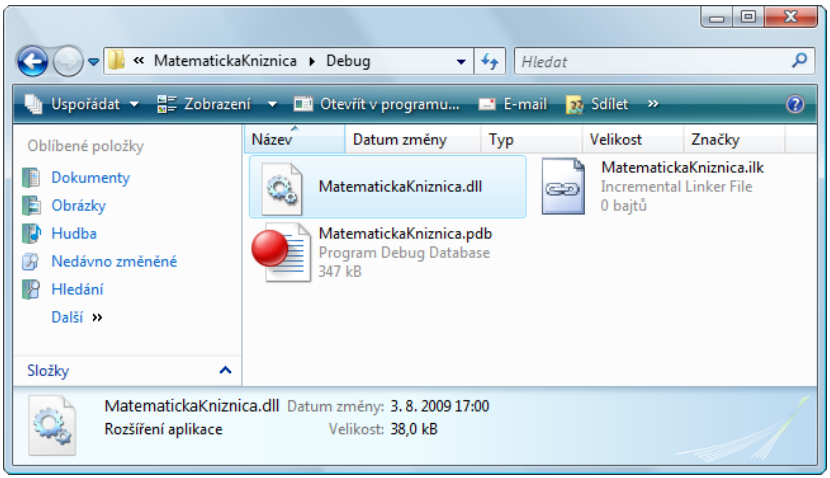


Obr. 17: Založenie novej knižnice tried v jazyku C++/CLI



Obr. 18: V knižnici tried sa nachádza trieda **SustavaLinearnychRovnic3X3**

3. Zostavíme knižnicu tried (**Build** → **Build Solution**). Po úspešnom zostavení bude vygenerovaný súbor MatematickaKniznica.dll, ktorý predstavuje našu knižnicu tried (obr. 19).

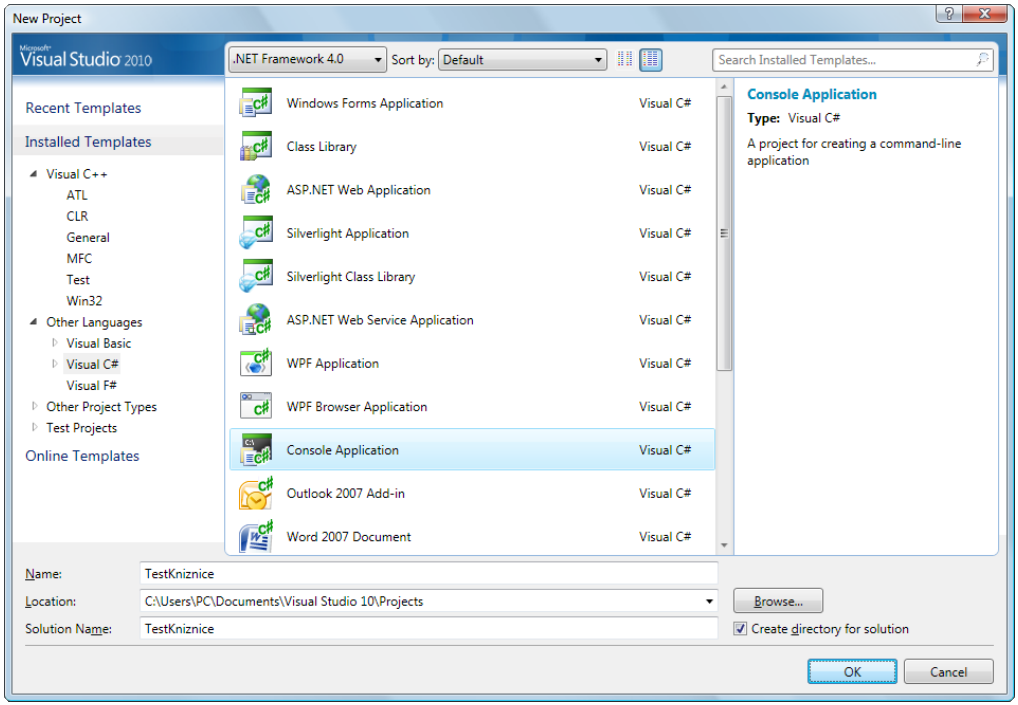


Obr. 19: Súbor s knižnicou tried (MatematickaKniznica.dll) jazyka C++/CLI

4. Zatvoríme projekt knižnice tried jazyka C++/CLI (**File** → **Close Solution**).

8.1.2 Vytvorenie štandardnej konzolovej aplikácie jazyka C# 4.0

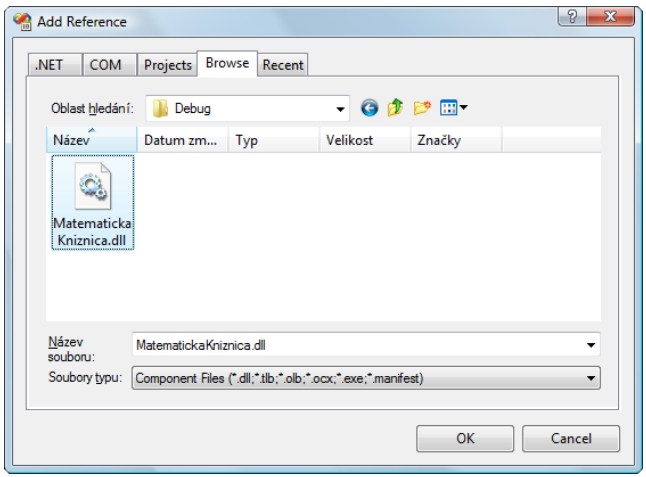
1. Založíme nový projekt štandardnej konzolovej aplikácie jazyka C# 4.0. V dialógovom okne **New Project** zvolíme projektovú šablónu **Console Application** jazyka C# 4.0 (**Other Languages** → **Visual C#** → **Console Application**). Projekt pomenujeme ako TestKniznice a aktivujeme tlačidlo **OK** (obr. 20).



Obr. 20: Založenie štandardnej konzolovej aplikácie jazyka C# 4.0

8.1.3 Pridanie odkazu na súbor s knižnicou tried jazyka C++/CLI

1. Otvoríme ponuku **Project** a klikneme na položku **Add Reference**.
2. V rovnomennom dialógovom okne vyberieme záložku **Browse**, vyhľadáme súbor s knižnicou tried jazyka C++/CLI (MatematickaKniznica.dll), a nakoniec klikneme na tlačidlo **OK** (obr. 21).



Obr. 21: Pridanie odkazu na knižnicu tried jazyka C++/CLI do projektu jazyka C# 4.0

Do projektu jazyka C# 4.0 sa pridá odkaz na knižnicu tried jazyka C++/CLI. Túto skutočnosť môžeme veľmi jednoducho overiť pohľadom do podokna **Solution Explorer**. Tu, pod uzlom **References**, pribudla položka **MatematickaKniznica**, ktorá reprezentuje knižnicu tried jazyka C++/CLI.

8.1.4 Efektívna interoperabilita medzi jazykmi C++/CLI a C# 4.0

1. Začneme využívať funkcionalitu knižnice tried jazyka C++/CLI priamo z jazyka C# 4.0:

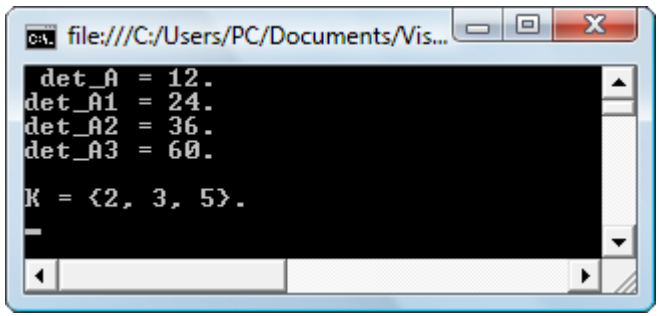
```
// Toto je zdrojový kód jazyka C# 4.0.
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
// Import menného priestoru, v ktorom je uložená deklarácia
// triedy SustavaLinearnychRovnic3X3.
using MatematickaKniznica;

namespace TestKniznice
{
    class Program
    {
```

```
        static void Main(string[] args)
        {
            // Inštanciácia triedy SustavaLinearnychRovnic3X3 a správna
            // inicializácia formálnych parametrov konštruktora.
            SustavaLinearnychRovnic3X3 sustava =
                new SustavaLinearnychRovnic3X3(
                    new int[,]
                    {
                        {2, -3, 1},
                        {1, 2, -1},
                        {2, 1, 1}
                    },
                    new int[]
                    {
                        0, 3, 12
                    }
            );

            // Volanie metódy za účelom automatického vyriešenia sústavy
            // 3 lineárnych rovníc s 3 neznámymi.
            sustava.Vyriesit();
            Console.Read();
        }
    }
}
```

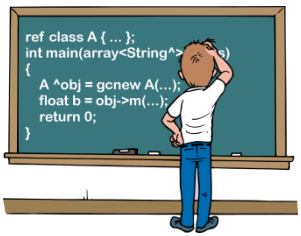
Komentár k zdrojovému kódu: Z rýdzo technického hľadiska nie je nutné, aby sme pomocou direktívy prekladača **using** zavádzali menný priestor **MatematickaKniznica**. Ak tak urobíme, nemusíme pri inštanciácii triedy **SustavaLinearnychRovnic3X3** uvádzať jej kvalifikovaný názov. To je istotne používateľsky prívetivé riešenie, pretože prispieva k tvorbe prehľadného a dobre štruktúrovaného zdrojového kódu. Ostatné úkony, najmä inštanciácia triedy a volanie metódy alokovanej inštancie, sú dobre pochopiteľné a neskrývajú žiadne kritické miesta. Zdrojový kód jazyka C# 4.0 rieši nasledujúcu sústavu 3 lineárnych rovníc s 3 neznámymi:

$$\begin{aligned} 2x_1 - 3x_2 + x_3 &= 0 \\ x_1 + 2x_2 - x_3 &= 3 \\ 2x_1 + x_2 + x_3 &= 12 \end{aligned}$$

Táto sústava má práve jedno riešenie, a síce $K = \{2, 3, 5\}$. Výsledok práce programu jazyka C# 4.0 môžeme vidieť na obr. 22.



Obr. 22: Výstup programu jazyka C# 4.0, ktorý využíva možnosti matematickej knižnice tried jazyka C++/CLI



9 Praktický príklad č. 5: Program uskutočňujúci paralelné grafické transformácie bitových máp

Vedomostná náročnosť:

Časová náročnosť: **20** minút

Keďže počet počítačov, ktoré sú osadené viacjadrovými procesormi od technologických spoločností AMD a Intel, sa za posledné roky signifikantne zvýšil, je úplne pochopiteľný aj nárast spotrebiteľského dopytu po paralelných programoch. Teda po programoch, ktoré sú schopné využiť všetku dostupnú výpočtovú kapacitu vysokovýkonných počítačových systémov. Aby mohli komerční vývojári úspešne čeliť tejto výzve, musia pri svojej práci začať uplatňovať paradigmu paralelného objektovo orientovaného programovania (POOP). Hoci analýza, návrh a implementácia paralelného programu predstavuje neľahkú úlohu, produktom tohto procesu je škálovateľný softvér, ktorý je optimalizovaný pre beh na počítačoch s viacjadrovými procesormi a na strojoch s viacerými procesormi.

V tomto praktickom cvičení vás zoznámime s ukážkou paralelného programovania v jazyku C++/CLI. Našou úlohou bude paralelizovať pôvodne sekvenčnú aplikáciu, ktorá vykonávala inverzné grafické transformácie množiny bitových máp.

Zdrojový kód sekvenčnej aplikácie, ktorá bola napísaná v jazyku C++/CLI, vyzeral takto:

```
#include "stdafx.h"

using namespace System;
using namespace System::Diagnostics;
using namespace System::Drawing;

// Deklarácia triedy, ktorá zapuzdruje funkcionalitu na výkon
// inverzných grafických transformácií.
ref class Bitmapa
{
private:
    Bitmap ^bitovaMapa;
    String ^subor;

public:
    Bitmapa(String ^subor)
    {
```

```
        bitovaMapa = gcnew Bitmap(subor);
        this->subor = subor;
    }
    // Metóda, ktorá spracúva inverziu bitovej mapy.
    void Invertovat()
    {
        int x, y;
        Color farba;
        for (x = 0; x < bitovaMapa->Width; x++)
        {
            for(y = 0; y < bitovaMapa->Height; y++)
            {
                farba = bitovaMapa->GetPixel(x, y);
                bitovaMapa->SetPixel(x, y,
                    Color::FromArgb(255 - farba.R, 255 - farba.G,
                    255 - farba.B));
            }
        }
        bitovaMapa->Save(subor->Insert(subor->Length - 4, "M"));
        delete bitovaMapa;
    }
};

int main(array<System::String ^> ^args)
{
    // Alokácia vektorového poľa 8 bitových máp.
    array<Bitmapa^> ^bitmapy = gcnew array<Bitmapa^>
    {
        gcnew Bitmapa("C:\\Obrazky\\obr_01.jpg"),
        gcnew Bitmapa("C:\\Obrazky\\obr_02.jpg"),
        gcnew Bitmapa("C:\\Obrazky\\obr_03.jpg"),
        gcnew Bitmapa("C:\\Obrazky\\obr_04.jpg"),
        gcnew Bitmapa("C:\\Obrazky\\obr_05.jpg"),
        gcnew Bitmapa("C:\\Obrazky\\obr_06.jpg"),
        gcnew Bitmapa("C:\\Obrazky\\obr_07.jpg"),
        gcnew Bitmapa("C:\\Obrazky\\obr_08.jpg")
    };
    Console::WriteLine("Prebiehajú sekvenčné inverzie bitových máp...");
    Stopwatch ^stopky = gcnew Stopwatch();
    stopky->Start();
    // Realizácia inverzných grafických transformácií všetkých bitových
    // máp uskladnených vo vektorovom poli.
    for (int i = 0; i < bitmapy->Length; i++)
    {
        bitmapy[i]->Invertovat();
    }
```

```
    stopky->Stop();

    Console::WriteLine("Sekvenčné inverzie bitových máp " +
        "sú hotové [celkový čas: {0} ms].", stopky->Elapsed.TotalMilliseconds);
    Console::Read();
    return 0;
}
```

Komentár k zdrojovému kódu: Inštancia deklarovanej triedy **Bitmapa** bude obsahovať obrazové body načítanej bitovej mapy. Cestu k požadovanej bitovej mape získa parametrický inštančný konštruktor, ktorý zabezpečuje alokáciu inštancie triedy **Bitmap** z menného priestoru **System.Drawing**. Keďže naša aplikácia je konzolová, je potrebné vložiť nielen odkaz na spomenutý menný priestor, ale tiež do projektu začleniť odkaz na zostavenie System.Drawing.dll. To urobíme nasledujúcim spôsobom:

1. Otvoríme ponuku **Project** a klikneme na položku **<NázovProjektu> Properties**.
2. Po zobrazení dialógového okna sa uistíme, že v stromovej štruktúre (ktorá je situovaná naľavo) je vybraná položka **Common Properties**.
3. Aktivujeme tlačidlo **Add New Reference**.
4. Len čo sa objaví rovnomenné dialógové okno, overíme, či je vybratá záložka **.NET**. Ak áno, v zozname **Component Name** vyhľadáme zostavenie System.Drawing.dll a vyberieme ho.
5. Klikneme na tlačidlo **OK**, čím pridáme odkaz na zostavenie System.Drawing.dll do projektu našej konzolovej aplikácie.
6. Stlačením tlačidla **OK** zatvoríme aj dialógové okno projektových vlastností.

Programový kód verejnej inštančnej a bezparametrickej metódy **Invertovat** sa sústreďuje na inverziu bitovej mapy. Pripomeňme, že inverzia je jednou z grafických transformácií, ktorá vypočítava opačnú (inverznú) farebnú informáciu každého obrazového bodu, z ktorých je bitová mapa zložená. V ďalšom texte budeme predpokladať, že zložky farebného vektora obrazového bodu môžu nadobúdať diskrétne celočíselné hodnoty z intervalu <0, 255>. Inverziou obrazového bodu P_1 s farebným vektorom $[R_1, G_1, B_1]$ získame obrazový bod P_2 s farebným vektorom $[255 - R_1, 255 - G_1, 255 - B_1]$. Keď metóda **Invertovat** dokončí inverziu, uloží upravenú bitovú mapu do nového rastrového súboru (názov tohto súboru vznikne z názvu pôvodného súboru, ku ktorému metóda pridá písmeno „M“). Nakoniec dochádza k volaniu metódy **Dispose** inštancie triedy **Bitmap**. Táto metóda je však volaná implicitne, a to prostredníctvom operátora **delete**.

V tele hlavnej metódy **main** alokujeme vektorové pole **bitmapy** pre 8 inštancií triedy **Bitmapa** (v každej inštancii tejto triedy bude pritom uložený rastrový obraz jednej bitovej mapy). Zdrojový kód predpokladá, že na pevnom disku C existuje priečinok Obrazky, v ktorom sa nachádza 8 bitových máp. Program vytvorí 8 inštancií triedy **Bitmapa**, pričom každá inštancia načíta jednu bitovú mapu zo spomenutého priečinka. Spracovanie programu pokračuje synchrónnym vykonaním inverzných grafických transformácií všetkých bitových máp. Aplikácia meria čas, ktorý je potrebný na dokončenie inverzie bitových máp.

Sekvenčnú aplikáciu sme podrobili výkonnostným testom. Pri testovaní sme použili nasledujúce dva počítače:

1. Notebook s 2-jadrovým procesorom AMD Turion 64 X2. Každé jadro tohto viacjadrového procesora je taktované na 2 GHz a disponuje 512 KB vyrovnávacej pamäte druhej úrovne (L2 cache).

2. Stolový počítač so 4-jadrovým procesorom Intel Core 2 Quad Q6600. Každé jadro tohto viacjadrového procesora je taktované na 2,4 GHz a obsahuje 2 MB L2 cache.

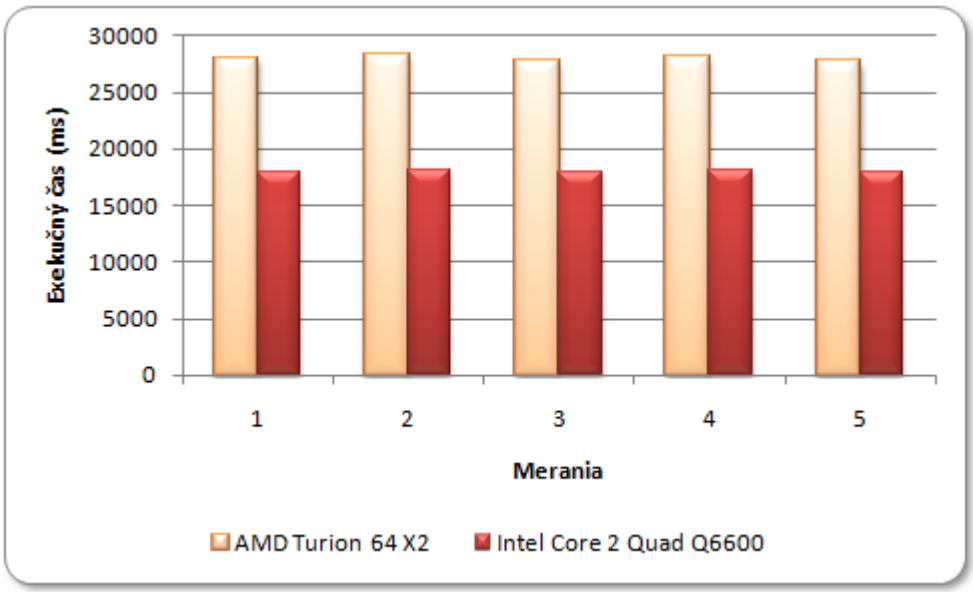
Namerané výsledky uvádzame v tab. 5.

Tab. 5: Výkonnosť sekvenčnej aplikácie

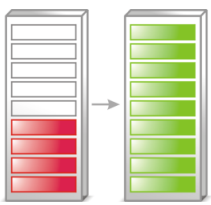
Výpočet	Exekučný čas (E_T) spracovania aplikácie na počítači s viacjadrovým procesorom (v ms)	
	AMD Turion 64 X2	Intel Core 2 Quad Q6600
1.	27979	17940
2.	28334	18085
3.	27741	17864
4.	27997	18120
5.	27663	17892
$\overline{E_T}$	**27943**	**17981**

Z uskutočnených testov vyplýva, že notebook s procesorom AMD Turion 64 X2 vykoná sekvenčné inverzie bitových máp za približne 28 sekúnd. Naproti tomu, stolový počítač s procesorom Intel Core 2 Quad Q6600 si s rovnakou úlohou poradí za 18 sekúnd.

Z porovnania výkonnosti oboch systémov je evidentné, že počítač vybavený 4-jadrovým procesorom je rýchlejší 1,5-krát.



Obr. 23: Výkonnosť sekvenčnej aplikácie na testovacích počítačoch



Paralelizácia pôvodne sekvenčnej aplikácie znamená toto: vytvoríme množinu pracovných programových vlákien a každé pracovné vlákno bude uskutočňovať inverznú grafickú transformáciu na jednej inštancii triedy **Bitmapa**. Syntaktický obraz paralelnej aplikácie jazyka C++/CLI je takýto:

```
int main(array<System::String ^> ^args)
{
    array<Bitmapa^> ^bitmapy = gcnew array<Bitmapa^>
    {
        gcnew Bitmapa("C:\\Obrazky\\obr_01.jpg"),
        gcnew Bitmapa("C:\\Obrazky\\obr_02.jpg"),
        gcnew Bitmapa("C:\\Obrazky\\obr_03.jpg"),
        gcnew Bitmapa("C:\\Obrazky\\obr_04.jpg"),
        gcnew Bitmapa("C:\\Obrazky\\obr_05.jpg"),
        gcnew Bitmapa("C:\\Obrazky\\obr_06.jpg"),
        gcnew Bitmapa("C:\\Obrazky\\obr_07.jpg"),
        gcnew Bitmapa("C:\\Obrazky\\obr_08.jpg")
```

```
    };
    // Konštrukcia vektorového poľa pracovných vlákien.
    array<Thread^> ^pracovneVlakna = gcnew array<Thread^>
    {
        gcnew Thread(gcnew ThreadStart(bitmapy[0], &Bitmapa::Invertovat)),
        gcnew Thread(gcnew ThreadStart(bitmapy[1], &Bitmapa::Invertovat)),
        gcnew Thread(gcnew ThreadStart(bitmapy[2], &Bitmapa::Invertovat)),
        gcnew Thread(gcnew ThreadStart(bitmapy[3], &Bitmapa::Invertovat)),
        gcnew Thread(gcnew ThreadStart(bitmapy[4], &Bitmapa::Invertovat)),
        gcnew Thread(gcnew ThreadStart(bitmapy[5], &Bitmapa::Invertovat)),
        gcnew Thread(gcnew ThreadStart(bitmapy[6], &Bitmapa::Invertovat)),
        gcnew Thread(gcnew ThreadStart(bitmapy[7], &Bitmapa::Invertovat))
    };
    Console::WriteLine("Prebiehajú paralelné inverzie bitových máp...");
    Stopwatch ^stopky = gcnew Stopwatch();
    stopky->Start();
    // Štart pracovných vlákien a spracovanie inverzných
    // grafických transformácií.
    for (int i = 0; i < pracovneVlakna->Length; i++)
    {
        pracovneVlakna[i]->Start();
    }
    // Primárne vlákno čaká, až kým všetky pracovné vlákna
    // nedokončia svoju činnosť.
    for (int i = 0; i < pracovneVlakna->Length; i++)
    {
        pracovneVlakna[i]->Join();
    }
    stopky->Stop();
    Console::WriteLine("Paralelné inverzie bitových máp " +
        "sú hotové [celkový čas: {0} ms].", stopky->Elapsed.TotalMilliseconds);
    Console::Read();
    return 0;
}
```

Komentár k zdrojovému kódu: Paralelizácia sekvenčnej aplikácie je vskutku intuitívna. Podstatou je explicitná tvorba poľa pracovných vlákien, pričom dbáme na to, aby sa počet pracovných vlákien zhodoval s počtom bitových máp, ktoré chceme podrobiť grafickej transformácii.

Upozornenie: Keďže plánujeme každé pracovné vlákno zviazať s inštančnou metódou objektu triedy **Bitmapa**, musíme parametrickému konštruktoru delegáta **ThreadStart** odovzdať nasledujúce dva argumenty:

1. odkaz na objekt triedy **Bitmapa**, s ktorým bude pracovné vlákno manipulovať,
2. adresu cieľovej inštančnej metódy, ktorá bude na pracovnom vlákne spracúvaná.

V tejto súvislosti je nutné podotknúť, že pri špecifikácii adresy cieľovej inštančnej metódy zadávame jej kvalifikovaný názov, pred ktorým sa nachádza unárny referenčný operátor (jeho prítomnosť je však len fakultatívna). Kvalifikovaný názov tvorí identifikátor triedy (**Bitmapa**), binárny rozlišovací operátor (::) a identifikátor inštančnej metódy (**Invertovat**).

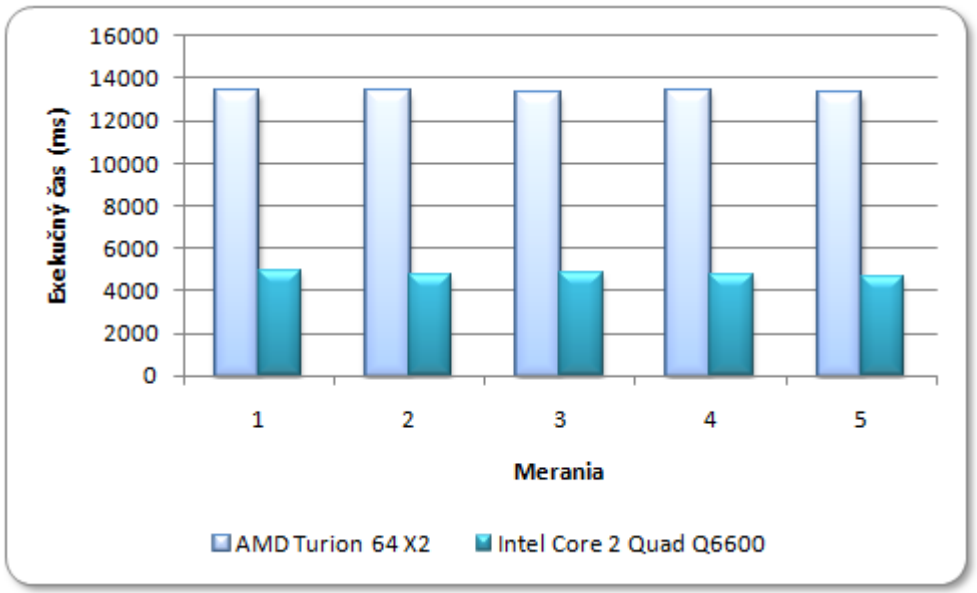
Každé pracovné vlákno necháme transformovať práve jednu bitovú mapu, čím dosiahneme priaznivé rozdelenie pracovného zaťaženia. Paralelná aplikácia využíva statický prístup, kedy v čase prekladu zdrojového kódu programu prijímame rozhodnutie o celkovom počte vytvorených pracovných vlákien. Ak by sme chceli zvýšiť citlivosť aplikácie, mohli by sme uprednostniť dynamický prístup, ktorý by nám dovolil zvýšiť mieru granularity pri tvorbe pracovných vlákien aplikácie.

Paralelnú aplikáciu sme rovnako otestovali. Výsledky našich meraní na testovacích počítačoch sú zhrnuté v tab. 6.

Tab. 6: Výkonnosť paralelnej aplikácie

Výpočet	Exekučný čas (E_T) spracovania aplikácie na počítači s viacjadrovým procesorom (v ms)	
	AMD Turion 64 X2	Intel Core 2 Quad Q6600
1.	13372	4923
2.	13355	4722
3.	13281	4836
4.	13407	4793
5.	13285	4694
$\overline{E_T}$	**13340**	**4794**

Po uskutočnených testoch môžeme konštatovať, že s paralelnou aplikáciou si počítače osadené viacjadrovými procesormi rozumejú veľmi dobre. Notebook s procesorom AMD Turion 64 X2 bol s inverznými grafickými transformáciami bitových máp hotový za približne 14 sekúnd. Na stolovom počítači s procesorom Intel Core 2 Quad Q6600 boli všetky operácie vykonané za menej ako 5 sekúnd.



Obr. 24: Výkonnosť paralelnej aplikácie na testovacích počítačoch

Zaujímavá bude istotne aj bližšia technická analýza výsledkov, ku ktorým sme sa počas testov dopracovali. Po prvé, stroj so 4-jadrovým procesorom je pri spracovaní paralelnej aplikácie rýchlejší ako počítač s 2-jadrovým procesorom. To je všeobecne platný princíp, ktorý dokazuje, že paralelná aplikácia je škálovateľná. Dokáže sa teda flexibilne prispôsobiť počítačom s viacerými exekučnými jadrami procesora. Samozrejme, úplnú dynamickú škálovateľnosť naša paralelná aplikácia neposkytuje, pretože sme počet pracovných vlákien fixne obmedzili na 8. Ak by sme paralelnú aplikáciu spustili na počítači s 8 exekučnými jadrami (napr. na počítači s dvomi 4-jadrovými procesormi), tak plánovací mechanizmus operačného systému by pridelil každému exekučnému jadru jedno pracovné vlákno (za týchto okolností by sa životné cykly pracovných vlákien nachádzali v tzv. ideálnych stavoch).

Po druhé, pomocou nasledujúceho matematického vzťahu kvantifikujeme nárast výkonnosti paralelnej aplikácie v porovnaní so sekvenčnou aplikáciou:

$$N_V = \frac{\bar{E}_{TS}}{\bar{E}_{TP}}$$

kde:

- N_V je nárast výkonnosti.
- $\bar{E}_{TS}$ je priemerný exekučný čas spracovania sekvenčnej aplikácie.
- $\bar{E}_{TP}$ je priemerný exekučný čas spracovania paralelnej aplikácie.

Nárast výkonnosti paralelnej aplikácie na počítači s procesorom AMD Turion 64 X2:

$$N_V = \frac{\bar{E}_{TS}}{\bar{E}_{TP}} = \frac{27943}{13340} = 2{,}09$$

Nárast výkonnosti paralelnej aplikácie na počítači s procesorom Intel Core 2 Quad Q6600:

$$N_V = \frac{\bar{E}_{TS}}{\bar{E}_{TP}} = \frac{17981}{4794} = 3{,}75$$

Počítačové vedy vravia, že nárast výkonnosti paralelnej aplikácie môže byť trojaký:

1. superlineárny: ak $N_V > n$,
2. lineárny: ak $N_V = n$,
3. sublineárny: ak $N_V < n$.

kde:

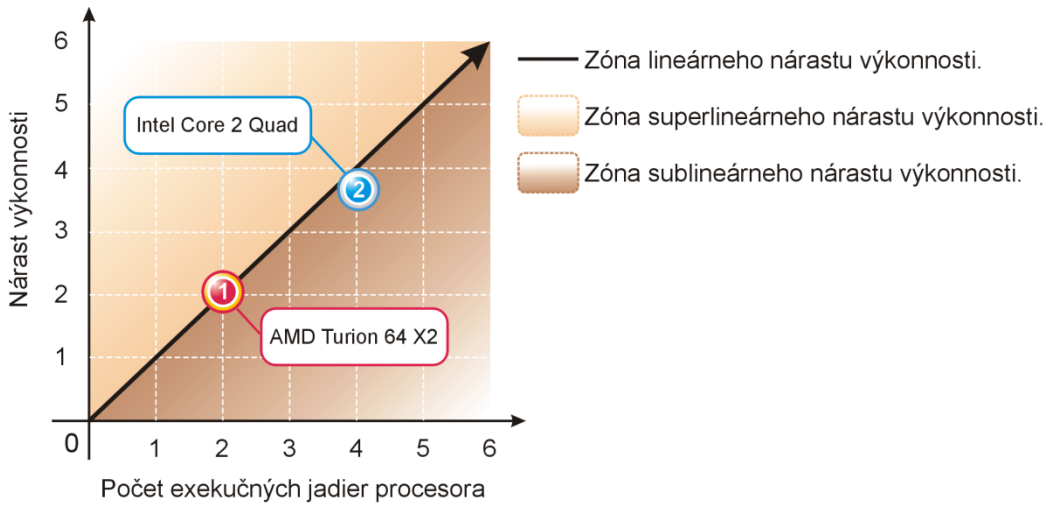
- n je počet exekučných jadier viacjadrového procesora.

Na počítači s 2-jadrovým procesorom by mala paralelná aplikácia byť dvakrát tak rýchla ako na uniprocesorovom počítači. Na stroji so 4-jadrovým procesorom by sme mali zaznamenať 4-násobný nárast výkonnosti. Za predpokladu, že sa paralelná aplikácia správa tak, ako sme uviedli v predchádzajúcich dvoch príkladoch, jej nárast výkonnosti je lineárny. Lineárny trend zvyšovania výkonnosti je v bežných praktických podmienkach ponímaný ako ideálny (a teda optimálny) stav. No je nutné podotknúť, že paralelná aplikácia oveľa častejšie

preukazuje sublineárny nárast svojej výkonnosti. V určitých (no bezpochyby vzácnejších) prípadoch je paralelná aplikácia schopná generovať aj superlineárny nárast výkonnosti.

Nárast výkonnosti paralelnej aplikácie na počítači s procesorom AMD Turion 64 X je 2,09. Z teoretického hľadiska je výkonnosť paralelnej aplikácie superlineárna, avšak keďže zdrojový kód aplikácie nepoužíva žiadne pokročilé optimalizačné techniky, tak po uskutočnení ďalšej súpravy testov by sme s najväčšou pravdepodobnosťou zaznamenali prepad výkonnosti paralelnej aplikácie do sublineárneho pásma.

Nárast výkonnosti paralelnej aplikácie na počítači s procesorom Intel Core 2 Quad Q6600 je 3,75. Výkonnosť aplikácie je tak sublineárna, a takú by sme v kontexte našej paralelnej aplikácie aj oprávnene očakávali.



Obr. 25: Výkonnostná mapa paralelnej aplikácie

O autorovi

Ing. Ján Hanák, MVP, vyštudoval Ekonomickú univerzitu v Bratislave. Tu, na Katedre aplikovanej informatiky Fakulty hospodárskej informatiky (KAI FHI), pracuje ako vysokoškolský pedagóg. Prednáša a vedie semináre programovania a vývoja počítačového softvéru v programovacích jazykoch C, C++ a C#. Okrem spomenutej trojice jazykov patrí k jeho obľúbeným programovacím prostriedkom tiež Visual Basic, C++/CLI a F# .

Je nadšeným autorom odbornej počítačovej literatúry. V jeho portfóliu môžete nájsť nasledujúce knižné tituly:

1. **C++/CLI – Praktické príklady**. Brno: Artax, 2009.
2. **C# 3.0 – Programování na platformě .NET 3.5**. Brno: Zoner Press, 2009.
3. **C++/CLI – Začíname programovat**. Brno: Artax, 2009.
4. **C#: Akademický výučbový kurz**. Bratislava: Vydavateľstvo EKONÓM, 2009.
5. **Základy paralelného programovania v jazyku C# 3.0**. Brno: Artax, 2009.
6. **Objektovo orientované programovanie v jazyku C# 3.0**. Brno: Artax, 2008.
7. **Inovácie v jazyku Visual Basic 2008**. Praha: Microsoft, 2008.
8. **Visual Basic 2008: Grafické transformácie a ich optimalizácie**. Bratislava: Microsoft Slovakia, 2008.
9. **Programovanie B – Zbierka prednášok (Učebná pomôcka na programovanie v jazyku C++)**. Bratislava: Vydavateľstvo EKONÓM, 2008.
10. **Programovanie A – Zbierka prednášok (Učebná pomôcka na programovanie v jazyku C)**. Bratislava: Vydavateľstvo EKONÓM, 2008.
11. **Expanzívne šablóny: Príručka pre tvorbu "code snippets" pre Visual Studio**. Bratislava: Microsoft Slovakia, 2008.
12. **Kryptografia: Príručka pre praktické odskúšanie symetrického šifrovania v .NET Framework-u**. Bratislava: Microsoft Slovakia, 2007.
13. **Príručka pre praktické odskúšanie vývoja nad Windows Mobile 6.0**. Bratislava: Microsoft Slovakia, 2007.
14. **Príručka pre praktické odskúšanie vývoja nad DirectX**. Bratislava: Microsoft Slovakia, 2007.
15. **Príručka pre praktické odskúšanie automatizácie aplikácií Microsoft Office 2007**. Bratislava: Microsoft Slovakia, 2007.

16. **Visual Basic 2005 pro pokročilé**. Brno: Zoner Press, 2006.
17. **C# – praktické příklady**. Praha: Grada Publishing, 2006.
18. **Programujeme v jazycích C++ s Managed Extensions a C++/CLI**. Praha: Microsoft, 2006.
19. **Přecházíme z jazyka Visual Basic 6.0 na jazyk Visual Basic 2005**. Praha: Microsoft, 2005.
20. **Visual Basic .NET 2003 – Začínáme programovat**. Praha: Grada Publishing, 2004.

V rokoch 2006 – 2009 bol jeho prínos vývojárskym komunitám ocenený celosvetovými vývojárskymi titulmi **Microsoft Most Valuable Professional (MVP)** s kompetenciou **Visual Developer – Visual C++**.

Ocenenia:

1. Spoločnosť Microsoft ČR udelila Ing. Jánovi Hanákovi, MVP, v roku 2009 ocenenie za napísanie prvej vysokoškolskej učebnice "C++/CLI - Začíname programovat" o algoritmizácii a programovaní v jazyku C++/CLI.

2. Spoločnosť Microsoft Slovakia udelila Ing. Jánovi Hanákovi, MVP, v roku 2009 ocenenie za nasadenie najnovších vývojárskych technológií do výučbového procesu v publikácii "Základy paralelného programovania v jazyku C# 3.0".

3. Spoločnosť Microsoft ČR udelila Ing. Jánovi Hanákovi, MVP, v roku 2009 ocenenie za mimoriadne úspešné odborné knižné publikácie "Objektovo orientované programovanie v jazyku C# 3.0" a "Inovácie v jazyku Visual Basic 2008".

4. Spoločnosť Microsoft Slovakia udelila Ing. Jánovi Hanákovi, MVP, v roku 2009 ocenenie za zlepšovanie akademického ekosystému a za signifikantné rozširovanie technológií a programovacích jazykov Microsoftu na akademickej pôde.

5. Spoločnosť Grada Publishing udelila knihe "C# - praktické příklady" Ing. Jána Hanáka v roku 2006 ocenenie "Najúspešnejšia novinka vydavateľstva Grada v oblasti programovania za rok 2006".

Kontakt s vývojármi a programátormi udržiava najmä prostredníctvom technických seminárov a odborných konferencií, na ktorých vystupuje. Za všetky vyberáme tieto:

- Konferencia **Software Developer 2007**, príspevok na tému „Představení produktu Visual C++ 2005 a jazyka C++/CLI“. Praha 19. 6. 2007.

- Technický seminár **Novinky vo Visual C++ 2005**. Microsoft Slovakia. Bratislava 3. 10. 2006.

- Technický seminár **Visual Basic 2005 a jeho cesta k Windows Vista**. Microsoft Slovakia. Bratislava 27. 4. 2006.

Ako autor má mnohoročné skúsenosti s prácou v elektronických a printových médiách. Počas svojej kariéry pôsobil ako odborný autor alebo odborný redaktor v nasledujúcich časopisoch: PC WORLD, SOFTWARE DEVELOPER, CONNECT!, COMPUTERWORLD, INFOWARE, PC REVUE a CHIP.

Dovedna publikoval viac ako 250 odborných a populárnych prác venovaných vývoju počítačového softvéru.

Akademický blog autora môžete sledovať na adrese: http://blog.aspnet.sk/hanja/.

Ak sa chcete s autorom spojiť, môžete využiť jeho adresu elektronickej pošty: hanja@stonline.sk.

Ing. Ján Hanák je Microsoft MVP (Most Valuable Professional – najcennejší odborník) s kompetenciou Visual Developer – Visual C++. Je autorom 20 odborných kníh, príručiek a praktických cvičení o programovaní a vývoji počítačového softvéru. Pracuje ako vysokoškolský pedagóg na Katedre aplikovanej informatiky Fakulty hospodárskej informatiky Ekonomickej univerzity v Bratislave. Prednáša a vedie semináre z programovania v jazykoch C, C++ a C#. V rámci svojej vedeckej činnosti sa zaoberá problematikou štruktúrovaného, objektovo orientovaného, komponentového, funkcionálneho a paralelného programovania.

ISBN: 978-80-87017-05-0