Microsoft

Ján Hanák

Inovácie v prostredí Visual Studio 2012

Windows 8

Visual Studio

Ján Hanák

Inovácie v prostredí Visual Studio 2012

(Príručka pre vývojárov, softvérových expertov a IT špecialistov)

Microsoft
2012

Autor: Ing. Ján Hanák, PhD., MVP

Inovácie v prostredí Visual Studio 2012
(Príručka pre vývojárov, softvérových expertov a IT špecialistov)

Recenzenti:	prof. Ing. Peter Závodný, PhD. Ing. Magdaléna Cárachová, PhD.
Vydanie:	prvé
Rok prvého vydania:	2012
Jazyková korektúra:	Ing. Ján Hanák, PhD., MVP
Vydal:	Artax, a.s., Žabovřeská 16, 616 00 Brno pre Microsoft Slovakia, s.r.o., Prievozská 4D, 821 09 Bratislava
Zaradenie titulu:	vysokoškolská učebnica
Cieľové publikum:	študenti informatiky na vysokých školách univerzitného typu, vývojári, programátori, softvéroví experti a IT profesionáli
ISBN:	978-80-87017-11-1

Obsah

Úvod

Príchod novej verzie integrovaného vývojového prostredia Visual Studio spoločnosti Microsoft je vždy veľkým sviatkom softvérových vývojárov, programátorov a IT profesionálov. Nové Visual Studio 2012 ponúka inovované programovacie jazyky Visual Basic 2012, C# 5.0 a C++ (konformné s najnovším ISO-štandardom C++11). Okrem vylepšených jazykových špecifikácií sa, samozrejme, zdokonalili aj prekladače, ladiace a spojovacie programy, rovnako ako aj ďalšie relevantné súčasti vývojového prostredia.

Produkt Visual Studio 2012 je dodávaný v troch základných verziách (*Professional*, *Premium* a *Ultimate*), ktoré sa odlišujú predovšetkým rozsahom poskytovanej funkcionality. Hoci „veľké" Visual Studio 2012 je zamerané najmä na profesionálnych vývojárov a počítačových odborníkov, spoločnosť Microsoft ani tento raz nezabudla na početné zástupy študentov a fanúšikov programovania, pre ktorých je určené odľahčené vývojové prostredie Visual Studio Express 2012. Expresný produkt je vyrábaný v troch príchutiach: *Visual Studio Express 2012 for Windows 8* (orientácia na vývoj WinRT-programov), *Visual Studio Express for Windows Desktop* (zacielenie na programovanie lokálnych aplikácií) a *Visual Studio Express 2012 for Web* (slúži na vývoj webovských aplikácií a distribuovaných programov).

Prostredie Visual Studio 2012 poskytuje vývojárom komplexnú podporu v procese analýzy, návrhu, implementácie, ladenia, optimalizácie a nasadenia moderných WinRT-programov bežiacich v systéme Windows 8.

Tip: Teraz môžu sofistikované WinRT-programy konštruovať vývojári pracujúci v programovacích jazykoch Visual Basic 2012, C# 5.0, C++ a JavaScript. Vďaka podpore vizuálneho návrhu v jazyku XAML je budovanie grafických rozhraní programov zacielených na programové rozhranie WinRT rýchle a efektívne.

Programátori v jazykoch Visual Basic 2012 a C# 5.0 môžu ťažiť z natívne implementovanej podpory pre paradigmu asynchrónneho programovania. Prostredníctvom novo zapracovaných kľúčových slov **Async** a **Await** (respektíve **async** a **await**) dokážu

programovať aplikácie, ktoré sú schopné vykonávať dosiaľ blokujúce operácie bez prerušenia ďalšieho toku programu. Fanúšikovia jazyka Visual Basic 2012 sa dočkali iterátorov, teda objektov umožňujúcich komfortnú programovú manipuláciu s objektmi, ktoré sú zoskupené v poliach a kolekciách.

Veľkú renesanciu však zažíva vo svojom natívnom vyhotovení jazyk C++. Prekladač produktu Visual C++ 2012 implementuje väčšinu nových rysov najnovšieho ISO-štandardu pre jazyk C++. K nim patria najmä lambda-výrazy, rozsahovo obmedzené cykly **for** a vylepšené knižnice pre jazyk C++ (štandardná knižnica jazyka C++, štandardná šablónová knižnica jazyka C++, aktívna šablónová knižnica jazyka C++ a knižnica PPL na paralelné programovanie v jazyku C++). Navyše, prekladač jazyka C++ teraz integruje dva užitočné optimalizačné stroje, a síce automatický vektorizátor a automatický paralelizátor, ktoré si poradia s implicitnou transformáciou existujúceho skalárneho sekvenčného zdrojového kódu na nový vektorovo-paralelný zdrojový kód. Využitie vektorových inštrukcií mikroprocesora a paralelné spracovanie viacerých tokov inštrukcií na viacerých výpočtových jadrách mikroprocesora znamená takmer vždy citeľný nárast výkonnosti vyvíjaného programu.

Softvéroví inžinieri spoločnosti Microsoft zapracovali do jazyka C++ nové komponentové rozšírenia, ktoré dovedna vytvárajú nový jazyk s označením C++/CX. Ten sa stáva užitočným prostriedkom pri zhotovovaní WinRT-programov, ktoré sú úplne konformné s novým vizuálnym rozhraním systému Windows 8 (takéto programy môžu byť priamo distribuované do elektronického obchodu Windows). Jazyk C++/CX dokazuje svoje kvality aj pri programovaní natívnych WinRT-komponentov, ktoré participujú v rôznych modeloch interoperability s riadenými komponentmi alebo riadenými programami. Počnúc produktom Visual C++ 2012 sa dá v jazyku C++ uplatniť aj paradigma heterogénneho paralelného programovania pomocou technológie C++ AMP (čiže akcelerovaného masívneho paralelizmu). Týmto spôsobom vedia vývojári rozložiť celkovú záťaž programu nielen na jednotlivé výpočtové jadrá centrálneho mikroprocesora (CPU), ale aj na výpočtové jadrá dedikovaného grafického mikroprocesora (GPU).

Cieľom tejto knihy je poskytnúť základný prehľad inovácií, ktoré boli v najnovšej verzii vývojového prostredia Visual Studio 2012 začlenené do programovacích jazykov Visual Basic 2012, C# 5.0 a C++.

Ján Hanák
Bratislava november 2012

Pre koho je určená táto kniha

Hlavným cieľovým publikom knihy sú softvéroví vývojári a programátori, ktorí pri svojej tvorivej práci používajú programovacie jazyky Visual Basic, C# a C++. Intenciou príručky je poskytnúť predstaviteľom cieľového publika hodnotný prehľadový kurz inovácií, noviniek a vylepšení, ktoré boli zapracované do spomínanej trojice programovacích jazykov v ich najnovších verziách.

Napriek tomu, že publikácia vykladá syntakticko-sémantické inovácie jazykov Visual Basic 2012, C# 5.0 a C++ s vysokou precíznosťou a s odporúčania hodnými vzdelávacími metodikami a štandardmi, orientuje sa hlavne na pokročilých programátorov. Práve táto skupina IT odborníkov dokáže z predkladanej knihy vyťažiť maximálnu pridanú hodnotu.

Tip: Ak by ste sa radi zoznámili s procesom vývoja WinRT-programov pre systém Windows 8 v programovacích jazykoch Visual Basic 2012, C# 5.0 a C++, uvádzame do pozornosti knihu „*Vývoj moderných WinRT-programov pre systém Windows 8*“, ktorá je zadarmo k dispozícii vo vývojárskej knižnici spoločnosti Microsoft situovanej na nasledujúce webovskej adrese: http://msdn.microsoft.com/sk-sk/dd727769.

V prípade, ak sa čitateľ necíti byť pokročilým vývojárom v jazykoch Visual Basic 2012, C# 5.0 a C++, radíme mu, aby najskôr venoval svoj čas štúdiu knižných zdrojov, ktoré sú primárne určené začínajúcim programátorom v zmienených programovacích jazykoch.

Obsahová štruktúra knihy

Kniha obsahuje tri kapitoly:

1. **Inovácie v jazyku Visual Basic 2012.** Táto kapitola opisuje novinky, ktoré prináša najnovšia verzia jazyka Visual Basic. Vývojári sa zoznámia s konštrukciou vlastných iterátorov a s ich praktickou aplikáciou pri implementácii iteratívnych algoritmov. Rovnako ovládnu teoreticko-praktické princípy a techniky paradigmy asynchrónneho programovania.

2. **Inovácie v jazyku C# 5.0.** Táto kapitola je určená vývojárom v jazyku C#. Jadrom výkladu je implementácia paradigmy asynchrónneho programovania v jazyku C# 5.0.

3. **Inovácie v jazyku C++ (podľa ISO štandardu C++11).** Táto kapitola je určená natívnym programátorom v jazyku C++. Tí sa v nej stretnú s lambda-výrazmi, automatickou vektorizáciou a automatickou paralelizáciou zdrojového kódu a tiež aj s komponentovými rozšíreniami jazyka C++, ktoré dovoľujú rýchlu stavbu programov kompatibilných s novým vizuálnym rozhraním systému Windows 8.

Z obsahovej štruktúry knihy je očividné, že dielo pokrýva informačné potreby viacerých skupín vývojárov. Preto predpokladáme, že používatelia knihy uplatnia najskôr selektívny režim štúdia, v ktorom budú vedení svojimi preferenciami v používaní toho-ktorého programovacieho jazyka. Pochopiteľne, vzdelávaciu hodnotu knihy možno absorbovať aj sekvenčne, teda po jednotlivých kapitolách počnúc tou prvou. Tento model však počíta s tým, že čitateľ ovláda všetky tri opisované programovacie jazyky (Visual Basic, C# a C++).

Typografické konvencie

Aby sme vám čítanie tejto knihy spríjemnili v čo možno najväčšej miere, prijali sme kódex typografických konvencií, pomocou ktorých došlo k štandardizácii a unifikácii použitých textových štýlov a grafických symbolov. Veríme, že prijaté konvencie napomôžu zvýšenie prehľadnosti a používateľskej prívetivosti výkladu. Prehľad použitých typografických konvencií uvádzame v tab. A.

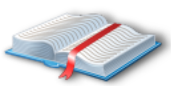
Tab. A: Prehľad použitých typografických konvencií

Typografická konvencia	Ukážka použitia typografickej konvencie
Štandardný text výkladu, ktorý neoznačuje zdrojový kód, identifikátory, modifikátory a kľúčové slová jazykov Visual Basic, C# a C++, ani názvy iných syntaktických elementov a entít, je formátovaný týmto typom písma.	Vývojová platforma Microsoft .NET Framework 4.5 vytvára spoločne s jazykmi Visual Basic, C# a C++ jednotnú technologickú bázu na vytváranie moderných riadených a natívnych programov v štýle WinRT.
Názvy ponúk, položiek ponúk, ovládacích prvkov, komponentov, dialógových okien, podporných softvérových nástrojov, typov projektov ako aj názvy ďalších súčastí grafického používateľského rozhrania sú formátované **tučným** písmom.	Projekt nového programu v štýle WinRT založíme v prostredí jazyka Visual Basic 2012 takto: 1. Na úvodnej stránke **Start Page** klepneme na položku **New Project**. 2. V dialógovom okne **New Project** rozvinieme v stromovej štruktúre **Templates** položku **Visual Basic** a klikneme na položku **Windows Store**. 3. Zo súpravy projektových šablón vyberieme šablónu **Grid App (XAML)**. 4. Do textového poľa **Name** zapíšeme názov nového projektu a stlačíme tlačidlo **OK**.

Symboly klávesov, klávesových skratiek a ich kombinácií sú uvádzané VEĽKÝMI PÍSMENAMI.	Ak chceme otvoriť už existujúci projekt jazyka Visual Basic 2012, použijeme klávesovú skratku CTRL+O.

Okrem typografických konvencií predstavených v tab. A sa môžeme na stránkach knihy stretnúť aj s informačnými ostrovčekmi, ktoré ponúkajú hodnotné informácie súvisiace s práve preberanou problematikou. Zoznam použitých informačných ostrovčekov a s nimi asociovaných grafických symbolov je zobrazený v tab. B.

Tab. B: Prehľad použitých informačných ostrovčekov

Grafický symbol informačného ostrovčeka	Názov informačného ostrovčeka	Charakteristika
	**Dôležité**	Upozorňuje čitateľov na dôležité skutočnosti, ktoré by mali mať v každom prípade na pamäti, pretože od nich môže závisieť pochopenie ďalších súvislostí alebo úspešné uskutočnenie postupu či pracovného algoritmu.
	**Poznámka**	Oboznamuje čitateľov s podrobnejšími informáciami, ktoré sa spájajú s vysvetľovanou problematikou. Hoci je miera dôležitosti tohto informačného ostrovčeka menšia ako pri jeho predchodcovi, vo všeobecnosti sa odporúča, aby čitatelia venovali doplňujúcim informačným vyhláseniam svoju pozornosť. Môžu sa tak dozvedieť nové fakty, alebo nájsť skryté súvislosti medzi už známymi poznatkami.
	**Tip**	Poukazuje na lepšie, rýchlejšie a efektívnejšie splnenie úlohy alebo postupu. Keď čitatelia uvidia v texte knihy tento informačný ostrovček, môžu si byť istí, že nájdu spôsob, ako produktívne dosiahnuť požadovaný cieľ.

Poďakovanie

Na tomto mieste by autor knihy rád vyjadril svoje poďakovanie recenzentom, prof. Ing. Petrovi Závodnému, PhD., a Ing. Magdaléne Cárachovej, PhD., za dôkladné posúdenie tejto publikácie a námety na jej ďalšie skvalitnenie. Rovnako úprimne ďakuje autor aj Mgr. Miroslavovi Kubovčíkovi, ktorý je kmeňovým členom vývojárskeho tímu slovenskej pobočky spoločnosti Microsoft, za výbornú niekoľkoročnú spoluprácu, ktorá priniesla vývojárskej komunite veľa hodnotných produktov.

Kapitola 1

Inovácie v jazyku Visual Basic 2012

Visual Studio

1 Inovácie v jazyku Visual Basic 2012

1.1 Iterátory

V programovaní charakterizujeme iterátor ako objekt, ktorý prechádza používateľom definovaným kontajnerom a uskutočňuje s prvkami tohto kontajnera naprogramované akcie. Spomínaným kontajnerom môže byť pole alebo kolekcia. V jazyku Visual Basic 2012 môžu vývojári vytvárať svoje vlastné iterátory, ktoré budú potom automaticky riadiť proces iteratívneho spracovania kolekcie prvkov prostredníctvom cyklu **For Each**.

Na syntaktickej úrovni je iterátor v jazyku Visual Basic 2012 reprezentovaný iterátorovou funkciou alebo iterátorovým blokom. Teraz preskúmame prvú možnosť, a teda prácu s iterátorovou funkciou.

Dôležité: Hoci syntakticky je v jazyku Visual Basic 2012 iterátor predstavovaný funkciou alebo blokom, skutočnosť je taká, že prekladač podľa použitej syntaktickej konštrukcie vždy zhotoví špeciálnu iterátorovú triedu, ktorú obdarí kompletnou funkcionalitou a zabezpečí, aby táto trieda implementovala požadované rozhrania.

Iterátorová funkcia je každá funkcia, ktorá vyhovuje nasledujúcim kritériám:

1. V hlavičke funkcie sa musí nachádzať modifikátor **Iterator**. Tento modifikátor je situovaný pred kľúčovým slovom **Function** a identifikátorom iterátorovej funkcie.

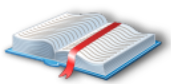
2. Funkcia môže disponovať variabilne mohutnou signatúrou. To znamená, že ako iterátorové funkcie sú podporované jednak bezparametrické a rovnako aj parametrické funkcie. Pokiaľ je funkcia parametrická, vstupné argumenty jej môžu byť odovzdané iba hodnotou. Povedané inými slovami, parametrická iterátorová funkcia smie definovať hodnotové formálne parametre (pomocou kľúčového

slova **ByVal**), avšak už nie odkazové formálne parametre (pomocou kľúčového slova **ByRef**).

3. Dátovým typom návratovej hodnoty iterátorovej funkcie musí byť negenerické alebo generické rozhranie platné pre enumerátory. V tomto smere prichádzajú do úvahy nasledujúce rozhrania: **IEnumerable**, **IEnumerable(Of T)**, **IEnumerator** a **IEnumerator(Of T)**. Funkcia teda vracia objekt, ktorý sa zrodil z triedy, ktorá implementuje konkrétne negenerické alebo generické rozhranie.

4. V tele funkcie sa vyskytuje príkaz **Yield**, ktorý vykonáva spracovanie a vrátenie práve jedného prvku kolekcie. Príkaz **Yield** sa skladá z kľúčového slova **Yield** a typovo silného výrazu, ktorý produkuje návratovú hodnotu funkcie.

Dôležité: Ak nie je určené inak, každá iterátorová funkcia je pomenovanou funkciou (disponuje svojím názvom, ktorý je explicitne uvedený pri definícii tejto funkcie). Radi by sme však podotkli, že ako iterátorová funkcia môže pôsobiť aj anonymná funkcia. Anonymné iterátorové funkcie bližšie rozoberieme v časti podkapitoly *1.1.1 Anonymné iterátorové funkcie*.

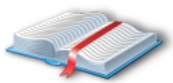


Poznámka: Iterátorová funkcia môže byť zaliata aj do tela triedy, prípadne môže vystupovať ako špeciálna prístupová metóda **Get** skalárnej inštančnej vlastnosti triedy. Z uvedeného vyplýva, že použitie modifikátora **Iterator** je povolené tiež v definícii vlastnosti triedy (**Property**).

Keďže sme vysvetlili základné syntaktické náležitosti iterátorovej funkcie, môžeme zostaviť presný algoritmus práce s iterátormi. Ten potom môžu vývojári upotrebiť v praxi pri návrhu a aplikácii vlastných iterátorov:

1. **Návrh a implementácia iterátorovej funkcie.** Iterátorová funkcia definuje fundamentálnu aplikačnú logiku iterátora, ktorý bude následne použitý na realizáciu vopred naprogramovaných akcií s kolekciou prvkov. (Pochopiteľne, definícia iterátorovej funkcie musí byť úplne konformná s normatívmi, ktoré sme už opísali.)

2. **Volanie iterátorovej funkcie prostredníctvom cyklu For Each.** Nový iterátor je aplikovaný v spojitosti s kolekciou prvkov, medzi ktorými iteruje cyklus **For Each**. Ako všetci programátori v jazyku Visual Basic 2012 dobre vedia, cyklus **For Each** prechádza všetkými prvkami kolekcie a umožňuje získavať ich kópie. Keď je cyklus **For Each** nanesený na kolekciu prvkov, ktorých spracovanie má na starosti novo dodaný iterátor, bude cyklom tento iterátor vždy (teda presnejšie pri každej iterácii cyklu) zavolaný.



Poznámka: Dodajme, že algoritmus operujúci s cyklom **For Each** a manipulujúci s prvkami kolekcie nie je schopný pozmeniť stavy týchto prvkov. To znamená, že pre potreby cyklu **For Each** má iterovaná kolekcia charakter konštantného objektu. Keď však preskúmame štýl práce cyklu **For Each** podrobnejšie, zistíme, že tento cyklus volá v asociácii s enumerátorom (ako objektom, ktorý implementuje rozhranie **IEnumerator** a ktorý je vystavený kolekciou) najskôr metódu **MoveNext**, a potom tiež skalárnu vlastnosť **Current**, ktorá vracia kópiu pôvodného prvku kolekcie. Takéto správanie je očakávané, pretože vlastnosť **Current** je určená iba na čítanie cieľového elementu (je teda **ReadOnly**).

3. **Spracovanie iterátorovej funkcie.** Vzhľadom na to, že iterátor vykonáva svoju činnosť s kolekciou prvkov, musí takáto kolekcia buď existovať, alebo musí byť vytvorená dynamicky, čiže za behu programu. Pripomeňme, že iterátor dokáže kooperovať s ktoroukoľvek kolekciou, ktorá implementuje negenerické (**IEnumerable**) alebo generické (**IEnumerable(Of T)**) rozhranie. V každom prípade, v algoritme, ktorý zavádza iterátorová funkcia, sa musí objavovať príkaz **Yield**, ktorý produkuje a vracia jeden prvok kolekcie. Príkaz **Yield** sa môže v tele iterátorovej funkcie vyskytovať raz alebo aj viackrát. Na tomto mieste je však dôležité podotknúť, že pri jednom volaní iterátorovej funkcie je príkaz **Yield** spracovaný práve jedenkrát s tým, že iterátor si poznačí aktuálnu pozíciu vo svojom algoritme a pri ďalšej exekúcii bude pokračovať tam, kde predtým skončil. Povedané formálnejšie, početnosť volania iterátorovej funkcie je v relácii typu 1:1 mapovaná na početnosť spracovania príkazov **Yield** v algoritme tejto funkcie.

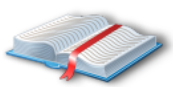
Nasledujúci výpis zdrojového kódu jazyka Visual Basic 2012 prezentuje program, ktorý manipuluje s novo navrhnutým iterátorom:

```
Module Module1
    Sub Main()
        'Aplikácia iterátora v cykle For Each.
        For Each i As Integer In Cisla()
            Console.WriteLine(i)
        Next
    End Sub
    'Definícia iterátorovej funkcie.
    Public Iterator Function Cisla() As IEnumerable
        Yield 1
        Yield 2
        Yield 3
    End Function
End Module
```

Komentár k programu jazyka Visual Basic 2012: Ako si môžeme všimnúť, iterátorová funkcia je koncipovaná ako pomenovaná bezparametrická funkcia s explicitne stanoveným dátovým typom svojej návratovej hodnoty. V tele iterátorovej funkcie sú umiestnené tri príkazy **Yield**, pričom každý z nich je spojený s celočíselnou konštantou. To znamená, že pri každom zavolaní iterátorovej funkcie sa do kolekcie pridá nový prvok, ktorého obsahom bude celočíselná konštanta.

Keď v hlavnej metóde **Main** programu dôjde k spracovaniu cyklu **For Each**, najskôr bude odštartovaná prvá iterácia tohto cyklu. Tá si vyžiada prvú aktiváciu iterátorovej funkcie. Iterátorová funkcia vykoná prvý príkaz **Yield**, ktorý vytvorí návratovú hodnotu tejto funkcie. Tou je prvok kolekcie s obsahom 1. Riadenie sa v tejto chvíli vracia späť cyklu **For Each**, ktorý získaný prvok dočasne uchová, pričom jeho textovú reprezentáciu odošle do výstupného dátového prúdu (pomocou zdieľanej metódy **WriteLine** triedy **Console**). Cyklus **For Each** začne druhú iteráciu, v ktorej opäť volá iterátorovú funkciu. Táto funkcia je aktivovaná už po druhýkrát, avšak jej spracovanie neprebieha znova od začiatku, ale tam, kde bolo predtým (čiže pri prvej aktivácii funkcie) prerušené. Keďže prvý príkaz **Yield** bol vykonaný, teraz, pri druhom zavolaní iterátorovej funkcie, bude spracovaný druhý príkaz **Yield**. Ten zostaví druhý prvok kolekcie a ponúkne ho cyklu **For Each**. Cyklus poskytnutý prvok spracuje a zavolá iterátorovú funkcie po tretí raz.

Vysvetlený kolobeh sa zase zopakuje, a potom sa skončí činnosť programu. Používateľ na výstupe preto uvidí postupnosť celočíselných konštánt (1, 2, 3). Sú to práve tieto konštanty, ktoré formujú prvky vytvorenej kolekcie, na ktorú bol aplikovaný naprogramovaný iterátor.



Poznámka: Zapracovanie iterátorov do jazyka Visual Basic 2012 so sebou prináša viacero syntakticky zaujímavých miest. Jedným z nich je determinácia iterátorovej funkcie v klauzule **In** cyklu **For Each**. Podvýraz **In Cisla()** v prípade predloženého fragmentu zdrojového kódu vyzerá inak, ako to, na čo sme zvyknutí pri bežnej práci s cyklom **For Each**. Keď manipulujeme s iterátormi, je dobré mať na pamäti skutočnosť, že výraz **Cisla()** je vyhodnotený ako volanie iterátorovej funkcie. Naša iterátorová funkcia je naprogramovaná tak, aby cyklu **For Each** postupne poskytovala práve jeden prvok dátovej kolekcie.

Ďalší program ukazuje, ako definovať parametrickú iterátorovú funkciu, ktorá dokáže vypočítať hmotnostné indexy skupiny ľudí:

```
Module Module1
    Sub Main()
        'Aplikácia iterátora v cykle For Each.
        For Each i As Integer In OsobyPreBMI(4)
            Console.WriteLine("Index BMI: {0}", i)
        Next
    End Sub
    'Definícia iterátorovej funkcie.
    Public Iterator Function OsobyPreBMI(ByVal pocetOsob As Integer) _
    As IEnumerable
        For osoba As Integer = 1 To pocetOsob
            Dim hmotnost As Integer, vyska As Single
            Console.Write("Zadajte hmotnost {0}. osoby (v kg): ", osoba)
            hmotnost = Convert.ToInt32(Console.ReadLine())
            Console.Write("Zadajte výšku {0}. osoby (v m): ", osoba)
            vyska = Convert.ToSingle(Console.ReadLine())
            Yield hmotnost / (vyska * vyska)
        Next
    End Function
End Module
```

Komentár k programu jazyka Visual Basic 2012: V signatúre iterátorovej funkcie (**OsobyPreBMI**) sa nachádza jeden formálny parameter determinujúci počet osôb,

ktorých indexy BMI by sme radi vypočítali. V tele iterátorovej funkcie analyzujeme jednotlivé osoby a získavame vstupné dáta. (Ako vieme, na výpočet hmotnostného indexu jednotlivca potrebujeme poznať jeho hmotnosť a výšku.) Za zmienku stojí pozícia príkazu **Yield** iterátora v tele cyklu **For**, ktorý je definovaný v iterátorovej funkcii. Nech nás neprekvapí fakt, že príkaz **Yield** je vnorený do tela cyklu **For**. Telo tohto cyklu totiž predstavuje pre príkaz **Yield** úplne platný programový kontext. Správanie príkazu **Yield** sa tým, samozrejme, nijako nemení, len to znamená, že prvok kolekcie pre cyklus **For Each** bude z tela iterátorovej funkcie vrátený po spracovaní každej iterácie cyklu **For**. Vďaka argumentu, ktorý programátor určí na strane klientskeho kódu, bude iterátorová funkcia vypočítavať hmotnostné indexy pre variabilný počet osôb.

1.1.1 Anonymné iterátorové funkcie

Hoci v bežnej programátorskej praxi sa budú vývojári v jazyku Visual Basic 2012 spoliehať skôr na pomenované iterátorové funkcie, je dobré vedieť, že prekladač tohto programovacieho prostriedku si rozumie aj s ich anonymnými ekvivalentmi. Štýl správania iterátora zostáva aj naďalej nemenný, na ten ostatne nemá existujúci alebo absentujúci identifikátor iterátorovej funkcie žiadny vplyv. Definícia anonymnej iterátorovej funkcie sa javí ako dobrá voľba predovšetkým vtedy, keď chceme navrhnúť, implementovať a aplikovať iterátor lokálne, avšak súčasne dynamicky.

Nasledujúca praktická ukážka demonštruje program, ktorý využíva funkcionalitu anonymného iterátora pri prehľadávaní priečinkov na cieľovom diskovom oddiele:

```
Imports System.IO
Module Module1
    Sub Main()
        'Definícia anonymnej iterátorovej funkcie.
        Dim priecinky = Iterator Function(disk As String) As IEnumerable
                            For Each priecinok In _
                                Directory.EnumerateDirectories(disk)
                                    Yield priecinok
                            Next
                        End Function
        For Each priecinok In priecinky("F:\")
```

```
            Console.WriteLine(priecinok)
        Next
    End Sub
End Module
```

Komentár k programu jazyka Visual Basic 2012: V momente, keď je našou intenciou tvorba anonymnej iterátorovej funkcie, nezadáme v hlavičke definície tejto funkcie žiaden identifikátor, len zapíšeme kontextovo sémanticky závislé kľúčové slová **Iterator** a **Function**. Anonymná iterátorová funkcia je v tomto prípade parametrická, pretože disponuje jedným formálnym parametrom typu **String**. Na druhej strane, anonymná iterátorová funkcia je rovnako funkciou s návratovou hodnotou (jej typom je rozhranie **IEnumerable**). Všetko, čo sa nachádza medzi príkazmi **Function** a **End Function**, tvorí telo iterátora. Po preštudovaní programovej ukážky vieme povedať, že v tele iterátorovej funkcie je zapísaný kód, ktorý iteratívne prechádza kolekciou priečinkov na vybratej diskovej partícii. Za prispenia príkazu **Yield** je iterátorom v každom priechode cyklu vrátená textová identifikácia jedného priečinka.

Pokiaľ v jazyku Visual Basic 2012 pracujeme s anonymnými metódami, je jasné, že niekde nablízku bude tiež mechanizmus typovej inferencie. To sa, koniec koncov, deje tiež v našom programe: kompletná definícia anonymnej iterátorovej funkcie totiž formuje inicializačný výraz, ktorého hodnota (tou je, zjednodušene povedané, odkaz na vygenerovanú anonymnú iterátorovú funkciu) je uložená do príslušnej odkazovej premennej s názvom **priecinky**. Definičnému výrazu tejto premennej chýba klauzula **As** s explicitnou špecifikáciou dátového typu. Ak typ premennej nezvolí programátor, musí tak urobiť prekladač prostredníctvom mechanizmu typovej inferencie.

Dôležité: Schopnosť prekladača jazyka Visual Basic 2012 inferovať dátové typy lokálnych entít je daná voľbou **Option Infer**. Za štandardných okolností je typová inferencia vždy aktívna (voľba **Option Infer** je zapnutá, a teda v stave **On**). Keby sa prekladač začal sťažovať na neschopnosť realizácie typovej inferencie, skontrolujeme, či je voľba **Option Infer** aktívna. Zapnúť ju môžeme buď vo vlastnostiach projektu (**Project** → **Properties**), alebo manuálne, zapísaním príkazu **Option Infer On** na prvý riadok zdrojového kódu práve otvoreného programového modulu.

Keď nasmerujeme kurzor myši na identifikátor odkazovej premennej **priecinky**, technológia IntelliSense nám povie, že jej implicitne zvoleným dátovým typom je **<Function(String) As System.Collections.IEnumerable>**. To je presvedčivý dôkaz o tom, že typom tejto premennej je anonymný delegát, ktorého signatúra sa úplne zhoduje so signatúrou anonymnej iterátorovej funkcie (z hľadiska typovej kompatibility ide o optimálny stav). Na úspešné spracovanie priraďovacieho príkazu musí prekladač uskutočniť tieto akcie:

1. Zostaviť deklaráciu anonymného delegáta a uskutočniť jeho inštanciáciu.
2. Vytvorený objekt anonymného delegáta inicializovať odkazom na cieľovú anonymnú iterátorovú funkciu. (Tým sa vytvorí silná väzba medzi objektom delegáta a iterátorom.)

Kedykoľvek bude objekt delegáta použitý, dôjde k (implicitne synchrónnej) aktivácii cieľovej iterátorovej funkcie. Táto situácia nastane v momente, keď bude prvýkrát spracovaný cyklus **For Each** v hlavnej metóde programu. Výstupom skúmaného programu bude textový zoznam priečinkov, ktoré sú alokované na príslušnom diskovom oddiele.

1.1.2 Iterátory a triedy kolekcií

Iterátory sú užitočnými nástrojmi najmä pri práci s triedami kolekcií. Nasledujúci fragment zdrojového kódu jazyka Visual Basic 2012 demonštruje triedu kolekcie s názvom **Procesory**. Táto trieda implementuje rozhranie **IEnumerable** a obsahuje tiež definíciu inštančnej iterátorovej metódy **GetEnumerator**, vďaka čomu môže klientsky kód elegantne manipulovať s kolekciou procesorov.

```
Module Module1
    'Deklarácia triedy kolekcie.
    Public Class Procesory
        Implements IEnumerable
        Private procesory() As String =
        {
            "Pentium", "Celeron", "Pentium III",
            "Pentium 4", "Core 2 Quad", "Core i7"
```

```
        }
        'Definícia iterátorovej metódy.
        Public Iterator Function GetEnumerator() As IEnumerator _
        Implements IEnumerable.GetEnumerator
            For i As Integer = 0 To procesory.Length - 1
                Yield procesory(i)
            Next
        End Function
    End Class
    Sub Main()
        Dim procesory As New Procesory()
        'Praktické použitie naprogramovaného iterátora.
        For Each procesor In procesory
            Console.WriteLine(procesor)
        Next
    End Sub
End Module
```

Komentár k zdrojovému kódu programu jazyka Visual Basic 2012: Trieda **Procesory** obsahuje tieto syntaktické súčasti:

- Dátový člen **procesory**, ktorý reprezentuje kolekciu šiestich procesorov firmy Intel. Dátový člen je okamžite inicializovaný pomocou syntaxe inicializátora kolekcie.

Tip: Inicializátor kolekcie je syntaktická konštrukcia, ktorá umožňuje promptne inicializovať polia a kolekcie priamym zadaním zoznamu inicializátorov do zložených zátvoriek (tie sú potom s premennou kolekcie spojené priraďovacím operátorom). To je prospešné, pretože týmto spôsobom sa efektívne vyhneme nutnosti inicializovať kolekciu viacnásobným volaním metódy **Add**.

- **Inštančná iterátorová metóda GetEnumerator.** Povinnosť zaviesť definíciu tejto metódy vyplýva triede z titulu implementácie rozhrania **IEnumerable**. V tele iterátorovej metódy prechádzame cyklom sekvenčne kolekciu procesorov a iterátorom vraciame postupne jeden procesor za druhým.

Zdrojový text, umiestnený v hlavnej metóde programu, je dobre čitateľný. Prvým krokom je vygenerovanie kolekcie. Druhým krokom je potom iterovanie naprieč elementmi už existujúcej kolekcie pomocou cyklu **For Each**.

1.2 Paradigma asynchrónneho programovania

Programovací jazyk Visual Basic 2012 uvádza spoločne s vývojovou platformou Microsoft .NET Framework 4.5 priamu podporu pre asynchrónne programovanie. Paradigma asynchrónneho programovania sa sústreďuje na implementáciu vzorcov asynchrónneho správania do pôvodne synchrónnych programov. Vo svete programovania sú synchrónne programy veľmi často ponímané ako sekvenčné programy, zatiaľ čo asynchrónne programy sú chápané ako paralelné programy. Táto interpretácia je celkom presná, aj keď medzi asynchrónnym a paralelným programovaním existujú isté rozdiely. Tak predovšetkým, primárnym cieľom paralelného programovania je implementácia paralelizmu do sekvenčného programu. Keď je zvolený a následne racionálne zavedený optimálny variant paralelizmu, spôsobí táto akcia maximalizáciu výkonu pôvodne sekvenčného softvéru. Je známe, že paralelný program je koncipovaný ako množina úloh, ktoré sú vykonávané súbežne na (najlepšie menej mohutnej) množine pracovných vlákien tohto programu. Pochopiteľne, celkový stupeň paralelizmu jednotlivých úloh je determinovaný ich vzájomnou dátovou a funkčnou koreláciou. Najvyšší stupeň asynchrónneho a v tomto kontexte tiež rýdzo paralelného spracovania je dosiahnutý pri nulovej alebo len minimálnej korelácii medzi riešenými úlohami. Keď chceme súbežne uskutočňovať úlohy v existujúcom sekvenčnom programe, aplikujeme naň paradigmu paralelného programovania, čím tento program pretransformujeme na ekvivalentný paralelný program.

Na druhej strane, fundamentálnym cieľom asynchrónneho programovania je maximalizácia miery elasticity voči používateľským vstupom. Ak do sekvenčného programu správne zavedieme asynchronizmus, takýto program sa razom premení z úplne synchrónneho programu na program asynchrónny. To znamená, že ktorákoľvek vetva, nachádzajúca sa v grafe, ktorý reprezentuje tok spracovania asynchrónneho programu,

nebude na signifikantne dlhý časový interval blokovaná žiadnou synchrónne realizovanou výpočtovou operáciou.

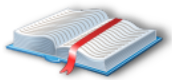
Pokúsme sa teraz porovnať štýly správania synchrónnych a asynchrónnych programov. Každý synchrónny program vykonáva svoje úlohy sekvenčne, a to vo vopred jasne danej sériovej postupnosti. Keď hlavná metóda synchrónneho programu (**Main**) zavolá cieľovú metódu (s názvom **MetodaA** a s návratovou hodnotou), uskutoční sa nasledujúci reťazec akcií:

1. **Spracovanie hlavnej metódy sa preruší, pričom dôjde k uloženiu jej aktuálneho kontextu.** Archivácia kontextu hlavnej metódy je významná operácia, pretože pri nej dochádza jednak k zaznamenaniu momentálneho stavu spracovania programu a rovnako aj k archivácii všetkých lokálnych a globálnych objektov, s ktorými hlavná metóda manipuluje.

2. **Synchrónne sa zavolá a spracuje cieľová metóda (MetodaA).** Riadenie je presunuté z hlavnej metódy programu na synchrónne vykonávanú cieľovú metódu. Majme na pamäti skutočnosť, že **MetodaA** si riadenie ponecháva v priebehu celého svojho spracovania a hlavnej metóde ho odovzdá až spoločne so svojou návratovou hodnotou. Čas spracovania tejto metódy je odvodený od zložitosti algoritmu, ktorý je implementovaný v jej tele. Vieme, že **MetodaA** disponuje svojou návratovou hodnotou – tá predstavuje výstupnú hodnotu (a teda výsledok práce) realizovaného algoritmu. Vo všeobecnosti sa dá len ťažko predpovedať, kedy **MetodaA** svoju návratovú hodnotu vygeneruje. Môže to trvať 10 milisekúnd, ale tiež napríklad 500 milisekúnd alebo dokonca 5 sekúnd.

3. **Uskutoční sa obnova kontextu hlavnej metódy a dôjde k spracovaniu výsledku činnosti synchrónne vykonanej cieľovej metódy.** Radi by sme zdôraznili, že volanie, spracovanie a návrat synchrónne aktivovanej cieľovej metódy sú všetko synchrónne operácie, ktoré generujú menšiu či väčšiu časovú latenciu. Bohužiaľ, hlavná metóda programu je blokovaná tak dlho, dokiaľ sa synchrónna **MetodaA** nevráti aj so svojím výsledkom. To nás privádza k poznaniu,

že hlavná metóda smie zavolať (opäť synchrónne) ďalšiu cieľovú metódu (s identifikátorom **MetodaB**) až potom, čo sa ukončí beh prvej cieľovej metódy.



Poznámka: V dôsledku časového oneskorenia, ktoré je ohraničené momentmi zavolania a návratu synchrónnej metódy, sa môže radikálne znížiť miera objektívnej i subjektívnej rýchlosti programu v očiach používateľa. V tomto smere platí priama úmera: čím väčší čas alokuje spracovanie synchrónnej metódy, tým viac je postrehnuteľná degradácia agility programu. Keď beží cieľová synchrónna metóda, používateľovi sa zdá, že program nič nerobí (pretože nereaguje na jeho ďalšie pokyny), a to aj napriek tomu, že to tak nie je (hlavná metóda programu čaká na riadenie, ktoré jej poskytne cieľová synchrónna metóda po svojom vrátení).

Najväčším problémom v štýle správania synchrónneho programu je blokovanie hlavnej metódy tohto programu. Hlavná metóda totiž pôsobí nielen ako manažér pracovných metód, ale tiež ako spracovateľ vstupných požiadaviek, ktoré generuje používateľ prostredníctvom periférnych zariadení počítača (najčastejšie ide o klávesnicu a myš, avšak do úvahy je potrebné vziať aj dotykové obrazovky, dynamicky pripojiteľné zariadenia cez USB-porty, tlačiarne, sieťové porty atď.).

Dôležité: Samozrejme, blokovanie, ktoré vyplýva zo synchrónne realizovaných metód, sa neobmedzuje iba na hlavnú metódu, ale vo všeobecnosti na akúkoľvek (zdrojovú) metódu, ktorá volá synchrónne ľubovoľnú (cieľovú) metódu.

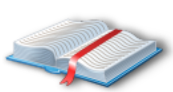
Riešenie problému zablokovania zdrojových metód vo vzťahu k cieľovým metódam spočíva v prepracovaní synchrónneho programu na program asynchrónny. To sa deje implementáciou asynchronizmu do pôvodne synchrónneho programu.

Pozrime sa, ako sa bude správať asynchrónny program, ktorý vznikne z existujúceho synchrónneho programu.

Pokiaľ hlavná metóda asynchrónneho programu (**Main**) zavolá asynchrónnu cieľovú metódu (**MetodaA**), stane sa toto:

1. Spracovanie hlavnej metódy sa preruší, pričom dôjde k uloženiu jej aktuálneho kontextu.
2. Asynchrónne sa zavolá cieľová metóda (**MetodaA**). Ak je cieľová metóda parametrická, uskutoční sa inicializácia jej formálnych parametrov.
3. Riadenie sa okamžite vráti hlavnej metóde, ktorá tak môže pokračovať vo svojej činnosti. Inými slovami, hlavná metóda je oslobodená od zablokovania a môže asynchrónne volať ďalšie cieľové metódy.
4. Cieľová metóda (**MetodaA**) je vykonaná asynchrónne, samozrejme bez toho, aby akokoľvek blokovala ďalšie spracovanie hlavnej metódy.
5. Keď asynchrónne spracúvaná **MetodaA** vráti výsledok svojej činnosti, hlavná metóda ho prijme, uloží a prípadne použije v nadchádzajúcich výpočtových procesoch.

Ako asynchrónny program v zásade vymedzujeme program, ktorý realizuje aspoň jednu akciu asynchrónne. Čím viac akcií uskutočňuje program asynchrónne, tým vyšší je stupeň asynchronizmu, ktorý bol do programu zavedený. Za štandardných okolností sa programátori snažia ako asynchrónne programovať operácie, ktoré by inak zapríčiňovali citeľné (z pohľadu používateľa) zablokovanie behu programu. Už sme konštatovali, že cieľom asynchrónneho programovania je maximalizovať mieru citlivosti voči používateľským vstupom. To príde vhod najmä pri návrhu programov s vizuálnym rozhraním (či už pomocou technológie WinForms alebo WPF), pri ktorých sme schopní uvoľniť ruky primárnemu vláknu programu (ktoré je zodpovedné za náležité ošetrenie vstupov používateľa).



Poznámka: Implementácia asynchrónneho programovania bola sprvu projektom technologického inkubátora spoločnosti Microsoft. Keď sa inkubačný projekt dostal do svojej finálnej etapy, Microsoft ho uvoľnil ako technologickú nadstavbu nad predchádzajúcou vývojovou platformou .NET Framework 4.0 a prisúdil jej označenie Async. V súčasnosti je technológia Async plnohodnotnou súčasťou platformy Microsoft .NET Framework 4.5. V záujme komfortnejšej tvorby asynchrónnych programových konštrukcií rozšírili

softvéroví inžinieri firmy Microsoft aj jazykové špecifikácie pre jazyky Visual Basic 2012 a C# 5.0. Zatiaľ čo v jazyku Visual Basic 2012 môžu vývojári využívať kľúčové slová **Async** a **Await**, pre vývojárov v jazyku C# 5.0 sú k dispozícii kľúčové slová **async** a **await**.

Praktických vývojárov bude istotne zaujímať to, aké typy úloh je lepšie písať asynchrónne. Podľa odporúčaní spoločnosti Microsoft je racionálne asynchrónne naprogramovať každú operáciu, ktorej spracovanie trvá dlhšie ako 50 milisekúnd. Naplnenie tohto odporúčania má vysokú prioritu, pretože už v samotnej bázovej knižnici tried nájdeme veľa adeptov metód, ktoré sú dostupné výhradne v asynchrónnej implementácii. Vývojári spoločnosti Microsoft totiž vykonali hĺbkovú analýzu pôvodných synchrónnych metód a tie menej výkonné a flexibilné nahradili ich asynchrónnymi verziami. Motív tohto konania je riadený heslom „fast & fluid", teda aby práca s programom bola rýchla a plynulá (a prostá akýchkoľvek blokovaní, s ktorými sme boli veľmi často konfrontovaní v synchrónnom kóde). Navyše, asynchrónne programovanie naberá na dôležitosti aj pri vývoji programov pre nové používateľské rozhranie systému Windows 8.

Vo všeobecnosti môžeme vyhlásiť, že kandidátmi na asynchrónne operácie sú všetky akcie, ktoré generujú zjavné časové oneskorenie. Konkrétne sa to týka hlavne vstupno-výstupných operácií (čítanie dát zo súboru, ukladanie dát do súboru), získavanie dát zo siete či generovanie dát zložitým algoritmom, ktorého exekučný čas prekračuje stanovenú hranicu 50 ms.

1.2.1 Praktické riešenie: Asynchrónne načítanie dát z fyzického súboru

V tomto tematickom celku prejdeme procesom zostrojenia praktického riešenia asynchrónneho programovania. Vyvinuté praktické riešenie bude oplývať nasledujúcimi charakteristikami:

- Pôjde o program s grafickým používateľským rozhraním, ktorý bude navrhnutý pomocou knižnice WinForms.

- Program najskôr (za asistencie objektového dátového prúdu) otvorí používateľom špecifikovaný textový súbor, a potom z neho asynchrónne načíta textovú správu. Táto správa bude umiestnená do vopred pripravenej vyrovnávacej pamäte. Predpokladáme, že načítané dáta textovej správy budú kódované v znakovej súprave Unicode.

- Program analyzuje načítanú textovú správu a zistí celkový počet znakov, ktoré táto správa obsahuje.

Stavba grafického používateľského rozhrania programu je vskutku priamočiara, pretože stačí, aby na hlavnom formulári bolo prítomné iba jedno tlačidlo. Len čo používateľ stisne toto tlačidlo, otvorí sa cieľový textový súbor, asynchrónne sa z neho načíta textová správa a tá bude vzápätí podrobená hĺbkovej analýze.

Najdôležitejším bodom pre vývojárov je zvládnutie procesu implementácie asynchronizmu do pôvodne synchrónneho programu. Proces zavedenia asynchrónneho spracovania môžeme v prostredí programovacieho jazyka Visual Basic 2012 a vývojovej platformy Microsoft .NET Framework 4.5 opísať nasledujúcou metodikou:

1. Metódu, ktorú budeme chcieť spracúvať asynchrónne, vybavíme modifikátorom **Async**. Tento modifikátor sa vyskytuje v hlavičke metódy, a to spravidla hneď za určením jej prístupových práv. Samotná prítomnosť modifikátora **Async** dáva prekladaču jasný signál o tom, že budeme chcieť, aby bola táto metóda vykonaná asynchrónne. Avšak pozor, prezencia modifikátora **Async** v hlavičke metódy ešte nerobí z pôvodnej synchrónnej metódy metódu asynchrónnu. V záujme skutočne asynchrónneho spracovania metódy je teda nutné uskutočniť ďalšie akcie.

Tip: Kvôli lepšiemu odlíšeniu asynchrónnych metód od tých synchrónnych bola prijatá nová pomenovacia konvencia, podľa ktorej sa identifikátor každej asynchrónnej metódy končí slovíčkom „Async“. Softvéroví vývojári spoločnosti Microsoft túto pomenovaciu konvenciu už v plnom rozsahu uplatnili pri návrhu architektúry bázovej knižnice tried (BCL) platformy .NET Framework 4.5. Pochopiteľne, firma Microsoft odporúča všetkým vývojárom, aby boli

konformní so spomenutou pomenovacou konvenciou vo všetkých vytvorených asynchrónnych programoch.

2. Asynchrónna metóda môže vystupovať buď ako procedúra **Sub**, alebo ako funkcia (**Function**). Prvá alternatíva je obvyklá vtedy, keď je asynchrónna metóda spracovateľom udalosti určitého objektu (napríklad udalosti **Click** tlačidla, ako ukážeme v chystanom praktickom riešení). Druhý variant je v praxi oveľa častejší: Ak je asynchrónnou metódou funkcia, zvyčajne ide o funkciu, ktorej návratovou hodnotou je objekt generickej triedy **Task(Of TResult)**. Pre úplnosť poznamenajme, že návratovou hodnotou funkcie môže byť tiež objekt negenerickej triedy **Task**. V obidvoch prípadoch však na objekty tried **Task** a **Task(Of TResult)** nahliadame ako na asynchrónne vykonávané úlohy. To je veľmi dôležité, pretože práve asynchrónne úlohy sú jadrom implementácie asynchronizmu do programu. Tieto úlohy sú spracúvané súbežne s hlavným tokom programu, pričom ho nikdy neblokujú.

3. Tok asynchrónnej metódy (s modifikátorom **Async** v jej hlavičke) môže byť prerušený vyslaním signálu so žiadosťou o vykonanie asynchrónnej úlohy. Syntakticky tento signál vysielame pomocou nového operátora **Await**. Tento operátor je aplikovaný na úlohu, ktorá má byť vykonaná asynchrónne. Po spracovaní príkazu s operátorom **Await** dochádza k naštartovaniu realizácie asynchrónnej úlohy s tým, že aktuálny beh asynchrónnej metódy sa preruší a riadenie sa vráti späť klientskemu kódu. To je prospešné z viacerých dôvodov. Po prvé, je naplánované spustenie asynchrónnej úlohy. Po druhé, riadenie je odovzdané volajúcej metóde, čo je stav, ktorý maximalizuje elasticitu v asynchrónnom komunikačnom modeli. A po tretie, po začatí spracovania asynchrónnej úlohy sa asynchrónna metóda ako taká ocitne v suspendovanom stave. Zmena stavovej charakteristiky asynchrónnej metódy je rozumné riešenie, pretože táto metóda čaká na dokončenie spracovania asynchrónnej úlohy. Celkový čas, ktorý bude nutné alokovať na realizáciu asynchrónnej úlohy, môže byť variabilne dlhý. To je však v poriadku, pretože beh asynchrónnej metódy a jej úlohy nepredstavuje úzke hrdlo programu.

4. Keď sa spracovanie asynchrónnej úlohy dokončí, obnoví sa beh asynchrónnej metódy, ktorá poskytne výsledok činnosti asynchrónnej úlohy klientskemu kódu.

Ďalej preskúmame kompletný syntaktický skelet praktického riešenia asynchrónneho programu:

```
'Import potrebných menných priestorov.
Imports System.IO
Imports System.Text

Public Class Form1

    'Definícia spracovateľa udalosti ako asynchrónnej metódy.
    Private Async Sub btnCitatDataAsync_Click(sender As Object, _
    e As EventArgs) Handles btnCitatDataAsync.Click
        Me.Text = "Prebieha asynchrónna operácia..."
        'Zavolanie asynchrónnej úlohy a spracovanie jej výsledku.
        Dim vysledokUlohy As Long = _
            Await AnalyzovatDataAsync(Environment.GetFolderPath( _
                Environment.SpecialFolder.Desktop) & "\Kniha.txt")
        Me.Text = "Výsledok asynchrónnej úlohy - počet znakov: " & _
            vysledokUlohy & "."
    End Sub

    'Definícia pracovnej asynchrónnej metódy.
    Public Async Function AnalyzovatDataAsync(subor As String) _
    As Task(Of Long)
        'Vytvorenie objektu na reprezentáciu znakov v súprave Unicode.
        Dim kodovanie As New UnicodeEncoding()
        'Spojenie súboru s dátovým prúdom.
        Dim prud As FileStream = File.Open(subor, FileMode.Open)
        'Alokácia vyrovnávacej pamäte.
        Dim vyrovnavaciaPamat() As Byte = New Byte(CInt(prud.Length)) {}
        'Asynchrónne načítanie textovej správy do vyrovnávacej pamäte.
        Await prud.ReadAsync(vyrovnavaciaPamat, 0, CInt(prud.Length))
        'Odpojenie dátového prúdu od súboru a dealokácia zdrojov.
        prud.Close()
        'Generovanie návratovej hodnoty pracovnej asynchrónnej metódy.
        Return kodovanie.GetString(vyrovnavaciaPamat).LongCount
    End Function
End Class
```

Komentár k zdrojovému kódu jazyka Visual Basic 2012: V triede primárneho formulára sa objavujú definície dvoch asynchrónnych metód. Zatiaľ čo prvou asynchrónnou

metódou je spracovateľ udalosti **Click** tlačidla s názvom **btnCitatDataAsync_Click**, druhá asynchrónna metóda je pracovná a jej pomenovanie znie **AnalyzovatDataAsync**. Asynchrónny spracovateľ udalosti generuje novú asynchrónnu úlohu, štartuje ju a nakoniec tiež čaká na jej výsledok. Pracovná asynchrónna metóda **AnalyzovatDataAsync** je parametrickou funkciou s návratovou hodnotou typu **Task(Of Long)**. Táto metóda vykonáva niekoľko akcií, no najvýznamnejšie sú tieto štyri:

1. Vytvorenie dátového prúdu a jeho asociácia s textovým súborom.
2. Alokácia vyrovnávacej pamäte na načítanie dát textovej správy.
3. Asynchrónna aktivácia metódy **ReadAsync** na dátovom prúde spojenom s textovým súborom. Je to práve metóda **ReadAsync**, ktorá je zodpovedná za asynchrónnu realizáciu vstupnej dátovej operácie. Načítanie dát z textového súboru sa preto uskutočňuje asynchrónne a postupne sa tiež zapĺňa pripravená vyrovnávacia pamäť.
4. Spočítanie znakov textovej správy a propagácia tohto kvantitatívneho ukazovateľa volajúcej metóde.

Vzhľadom na to, že pracovná metóda **AnalyzovatDataAsync** je asynchrónna, môžeme v jej tele pracovať s operátorom **Await**. Prekladač jazyka Visual Basic 2012 implicitne očakáva, že sa v tele asynchrónnej metódy budú objavovať body prerušenia, ktoré sú determinované miestami, na ktorých bol aplikovaný operátor **Await**. To je koniec koncov oprávnená predikcia, pretože po použití operátora **Await** je vykonaná asynchrónna úloha. Ak prekladač nenájde v definícii asynchrónnej metódy výraz alebo príkaz s operátorom **Await**, generuje varovanie. Samozrejme, za týchto okolností nebude predmetná metóda vykonaná asynchrónne, ale synchrónne.

Aby program po svojom zostavení a spustení fungoval správne, očakávame, že na pracovnej ploche operačného systému sa nachádza rýdzo textový súbor s textovými dátami. Keď používateľ spustí program a klikne na tlačidlo, textové dáta budú zo súboru načítané asynchrónne. Podobne bude vykonaná aj ich analýza a celková dĺžka textovej správy sa nakoniec objaví v záhlaví hlavného formulára. Pri testovaní asynchrónneho programu odporúčame použiť čo najrobustnejší textový súbor. Čím je totiž obsah

textového súboru väčší, tým dlhšie bude trvať aj jeho načítanie. Nuž a práve v tomto čase vieme identifikovať výhody spojené s implementáciou asynchronizmu do programu. Napríklad, hlavný formulár môže byť voľne presúvaný, minimalizovaný alebo maximalizovaný. Žiadna z týchto akcií nespôsobí nečinnosť programu či chyby v prekresľovaní jeho grafického používateľského rozhrania.

Kapitola 2

Inovácie v jazyku C# 5.0

Visual Studio

2 Inovácie v jazyku C# 5.0

Hlavnou syntakticko-sémantickou inováciou programovacieho jazyka C# 5.0 je podpora paradigmy asynchrónneho programovania. Je to už istý čas, čo softvéroví vývojári spoločnosti Microsoft uplatňujú pri vývoji programovacích jazykov Visual Basic a C# stratégiu koevolúcie. Praktickou implikáciou tohto vyváženého prístupu je skutočnosť, že paradigma asynchrónneho programovania je v oboch spomenutých prostriedkoch implementovaná s rovnakou robustnosťou.

2.1 Synchrónne a asynchrónne výpočtové procesy

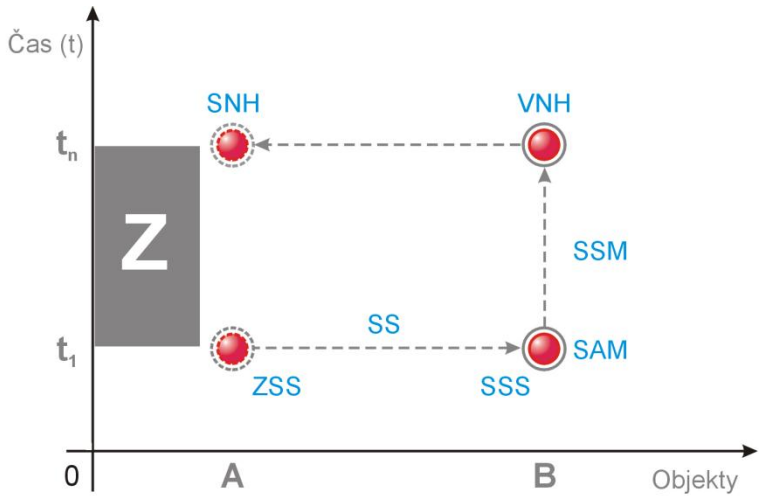
Za bežných okolností je výstupom prekladača jazyka C# 5.0 sekvenčný objektový program, ktorý realizuje sekvenčné výpočtové procesy. Na vyššej úrovni granularity je každý sekvenčný výpočtový proces tvorený z množiny sériovo spracúvaných výpočtových operácií. Sériové spracovanie znamená vykonanie výpočtových operácií v lineárnej postupnosti. Sekvenčný objektový program pracuje s objektmi, ktoré uskutočňujú vopred naprogramované akcie rovnako sekvenčne. Navyše, tiež modely, v ktorých tieto objekty prichádzajú do komunikačných väzieb, sú rýdzo sekvenčné. Znamená to, že jeden objekt (klient) je blokovaný dovtedy, kým mu druhý objekt (server) neposkytne požadovanú službu (napríklad výsledok sekvenčne realizovaného výpočtového procesu). Vždy, keď sa v sekvenčnom objektovom programe pri požiadavke o poskytnutie softvérovej služby stretávame s aspektom blokovania klienta serverom, môžeme konštatovať:

1. **Zúčastnené objekty (klient a server) komunikujú synchrónne.** Ak už raz klient vyšle požiadavku serveru, je znehybnený, pričom ďalšiu požiadavku (buď rovnakému, alebo inému serveru) je schopný vyslať až po dokončení synchrónne spracovaného výpočtového procesu prvotne dopytovaného servera.

2. **Synchrónny komunikačný a výpočtový model sú sekvenčnej povahy.** Synchrónne vykonávané akcie sú vykonávané sekvenčne, a nie paralelne. Tento

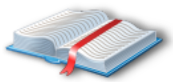
fakt obmedzuje jednak flexibilitu pracovných modelov objektov a jednak často pôsobí ako úzko hrdlo sekvenčného objektového programu.

Vizualizácia synchrónneho komunikačného a výpočtového modelu je znázornená na obr. 2.1.



Obr. 2.1: Synchrónny komunikačný a výpočtový model medzi objektmi

Na obr. 2.1 vidíme synchrónnu komunikáciu objektov **A** a **B**, pričom objekt **A** vystupuje v komunikačnom modeli ako klient, zatiaľ čo objekt **B** zohráva úlohu servera. Povedané menej formálne, objekt **A** bude generovať požiadavku na vykonanie cieľovej softvérovej služby, ktorá bude vzápätí odoslaná a explicitne spracovaná objektom **B**. Je všeobecne známe, že objekty v objektovo orientovanom systéme komunikujú prostredníctvom mechanizmu správ. Graf vraví, že v čase $\mathbf{t_1}$ objekt **A** vytvára a zasiela synchrónnu správu objektu **B** (značka **ZSS**). Synchrónna správa (**SS**) putuje k objektu **B**, ktorý ju v rovnakom čase prijíma (**SSS**).

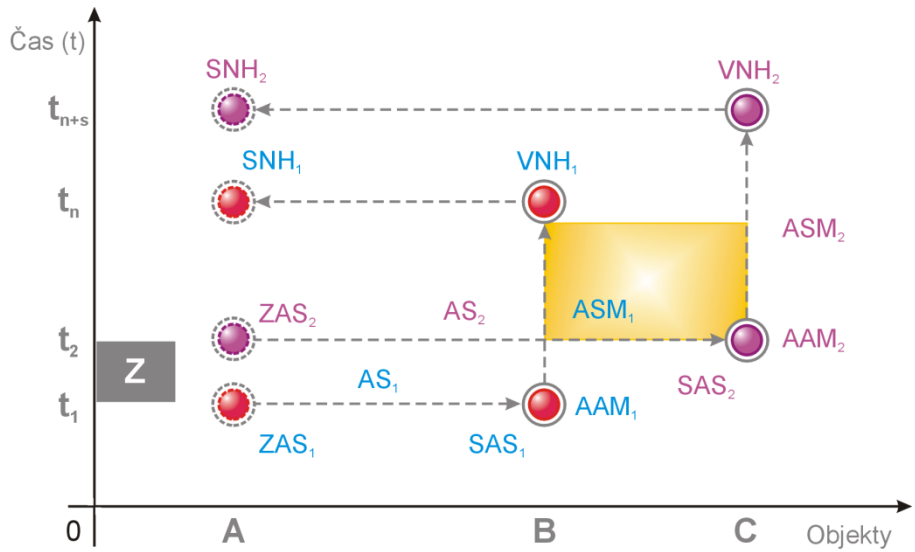


Poznámka: Samozrejme, v reálnom objektovom systéme sa časy zaslania a doručenia synchrónnej správy odlišujú. Hoci istá časová latencia tu vzniká, v našom modeli predpokladáme, že je veľmi malá, a preto si dovoľujeme ju zanedbať.

Keď server akceptuje synchrónne doručenú správu od klienta, konfrontuje ju so správami uloženými vo svojom protokole správ. Protokol správ je dátová štruktúra, ktorá zabezpečuje jednoznačné bijektívne mapovanie správ na cieľové metódy, ktoré budú vyvolané ako reakcie na doručenie správ. Často hovoríme, že na správy uložené v protokole správ je objekt citlivý. Jednou z anticipácií predostretého synchrónneho komunikačného a výpočtového modelu je premisa, že server je citlivý na správu, ktorú mu klient pošle. V tom momente server synchrónne aktivuje cieľovú metódu (**SAM**). Práve táto metóda je zodpovedná za realizáciu synchrónneho výpočtového procesu. Synchrónne spracovanie metódy (**SSM**) si vyžaduje určitý strojový čas – v našom modeli je tento čas ohraničený intervalom $<\mathbf{t_1}, \mathbf{t_n}>$.

Je dôležité poznamenať, že počas synchrónneho spracovania metódy servera je klient blokovaný, pretože čaká na výsledok tejto metódy. Len čo je synchrónne realizovaný výpočtový proces dokončený, je klientovi poskytnutý jeho výsledok. Technicky ide o vrátenie návratovej hodnoty synchrónne vykonávanej metódy (**VNH**), ktorú v rovnakom čase klient spracuje (**SNH**). Je pochopiteľné, že čas blokovania klienta serverom je totožný s časom spracovania synchrónne vyvolanej metódy servera. Celkový čas blokovania označujeme ako priamu časovú závislosť klienta od servera a graficky ju znázorňujeme štvoruholníkom (**Z**) medzi bodmi $\mathbf{t_1}$ a $\mathbf{t_n}$ na vertikále grafu.

Pre triviálne udalostné objektové programy je synchrónny komunikačný a výpočtový model postačujúcim riešením. Na stavbu robustnejších objektových aplikácií s atribútmi ako je vysoká flexibilita a reaktibilita vo vzťahu k podnetom finálnych používateľov však musíme nasadiť silnejší aparát – asynchrónny komunikačný a výpočtový model (grafické znázornenie uvádzame na obr. 2.2).



Obr. 2.2: Asynchrónny komunikačný a výpočtový model medzi objektmi

V záujme demonštrácie nuáns asynchrónneho komunikačného a výpočtového modelu v objektovom programe budeme potrebovať aspoň tri objekty (**A**, **B**, **C**). Delegácia právomocí je v tejto situácii nasledujúca: objekt **A** pôsobí ako klient, objekt **B** vystupuje ako prvý server a objekt **C** reprezentuje druhý server. V čase $\mathbf{t_1}$ generuje klient **A** prvú asynchrónnu správu ($\mathbf{AS_1}$), ktorú zasiela prvému serveru **B** ($\mathbf{ZAS_1}$). Objekt **B** túto správu v rovnakom čase prijme a v nadväznosti na to aktivuje príslušnú asynchrónnu metódu ($\mathbf{AAM_1}$). Asynchrónne aktivovaná metóda realizuje asynchrónny výpočtový proces ($\mathbf{ASM_1}$).

Konkurenčnou výhodou implementovaného asynchronizmu je okamžité vrátenie riadenia späť klientovi – objektu **A.** Takto môže klient **A** už vo veľmi blízkom čase $\mathbf{t_2}$ generovať ďalšiu asynchrónnu správu ($\mathbf{AS_2}$) a túto zaslať druhému serveru **C**. Objekt **C** spracuje doručenú asynchrónnu správu ($\mathbf{SAS_2}$) a asynchrónne vyvolá potrebnú cieľovú metódu ($\mathbf{AAM_2}$). Aj v tejto chvíli sa kontrola vracia späť klientovi **A**, ktorý by mohol v najbližšom možnom čase (napríklad $\mathbf{t_3}$) skonštruovať novú správu a propagovať ju ďalšiemu z dostupných serverov (napríklad **D**). Celkový čas spracovania asynchrónne aktivovanej metódy servera **B** je determinovaný intervalom $<\mathbf{t_1}, \mathbf{t_n}>$. Analogicky, celkový čas

nevyhnutný na exekúciu asynchrónne vyvolanej metódy servera **C** je ohraničený intervalom $<\mathbf{t_2}, \mathbf{t_{n+s}}>$.

Ako zisťujeme, asynchronizmus v komunikačnom modeli medzi klientom a servermi zabezpečuje odblokovanie klienta a maximalizáciu miery jeho reaktibility voči serverom. Druhá cenná devíza spočíva v implementácii asynchronizmu do realizácie výpočtových procesov, ktoré predstavujú automatizáciu softvérových služieb garantovaných objektovými servermi. Tieto výpočtové procesy sú uskutočňované paralelne. V kontúrach predloženého grafu to presnejšie znamená, že asynchrónne aktivované metódy serverov **B** a **C** sú v časovom rozpätí $<\mathbf{t_2}, \mathbf{t_n}>$ vykonávané súbežne. (Toto časové rozpätie je v grafe reprezentované štvoruholníkom s oranžovou podkladovou farbou.) Čím väčší časový úsek sú metódy serverov vykonávané paralelne, tým vyšší je stupeň paralelizmu objektového programu. Prirodzene, cieľom vývojárov je pracovať v prekrytom čase, a teda snaha o maximálnu možnú súbežnosť paralelne realizovaných výpočtových procesov.

2.2 Aplikačné domény pre asynchrónne výpočtové procesy

Asynchrónne výpočtové procesy, ktoré sú naprogramované v telách asynchrónnych inštančných alebo statických metód, sa prakticky využívajú najmä v týchto problémových oblastiach:

1. **Dolovanie dát z externých a distribuovane rozptýlených dátových zdrojov.** Typickým príkladom je operácia získania dát zo vzdialeného webového servera. Vývojári vedia, že konkrétna rýchlosť vykonania tejto operácie je závislá od viacerých faktorov, ako je architektúra siete, priepustnosť siete, nutnosť realizácie šifrovacích a dešifrovacích techník, aktuálna vyťaženosť webového servera, atď. V každom prípade je rozumné očakávať existenciu priamej časovej latencie, ktorá ovplyvňuje mieru spokojnosti používateľa so softvérovou aplikáciou. Prístupy k externým a distribuovaným dátovým zdrojom preto vždy uskutočňujeme

asynchrónne. Tak dokážeme odblokovať všetkých klientov, ktorí sú závislí od správnej realizácie tohto asynchrónne vykonávaného výpočtového procesu.

2. **Realizácia vstupno-výstupných operácií.** Čítanie dát z fyzických dátových súborov, ich úprava a následná archivácia predstavujú operácie, ktoré pri implicitnej synchrónnej implementácii blokujú primárne programové vlákno. Tým narúšajú plynulosť správneho prekresľovania grafického rozhrania programu. Je známe, že pri načítavaní a ukladaní dát vzniká vždy istá časová latencia. Veľkosť tejto latencie rastie priamo úmerne s mohutnosťou množiny prenášaných dátových jednotiek. Aby sme sa vyhli obrazu zdanlivo neaktívnej aplikácie, nahrádzame pôvodné synchrónne vstupno-výstupné algoritmy ich asynchrónnymi ekvivalentmi.

3. **Spracovanie grafických transformácií.** Či už máme na mysli grafické operácie vykonávané v dvojrozmernom (2D) a trojrozmernom (3D) priestore, alebo uvažujeme o hladkom nanášaní animácií vo viacerých vrstvách, vždy zisťujeme, že použitie synchrónnych grafických algoritmov nás neprivedie k očakávanému výsledku. Asynchrónne spracovanie grafických transformácií (ku ktorým patrí napríklad translácia, rotácia, skosenie, či zmeny mierky) zvýši dynamiku programu a maximalizuje schopnosť rýchlej reakcie na podnety používateľa.

Pochopiteľne, kolekcia uvádzaných aplikačných domén pre nasadenie asynchronizmu nie je vyčerpávajúca. Vo všeobecnosti smieme vyhlásiť, že výpočtové procesy uskutočňujeme asynchrónny vždy, keď by blokovanie klientov bolo natoľko signifikantné, že by sa rapídne znížila ich miera reaktibility.

2.3 Konštrukcia asynchrónnych metód v jazyku C# 5.0

Konkrétna implementácia asynchronizmu je v jazyku C# 5.0 determinovaná aplikáciou návrhového vzoru úlohovo orientovaného asynchronizmu (angl. *Task-based Asynchronous*

Pattern, TAP). Na rozdiel od predchádzajúcich prístupov nie sú nutné dve metódy (bežne označované ako **BeginInvoke** a **EndInvoke**), ale len jedna asynchrónna metóda, ktorá obsahuje asynchrónny algoritmus. Predpokladáme, že asynchrónna metóda bude vykonávať konečnú a neprázdnu množinu asynchrónnych úloh. Asynchrónnu úlohu budeme chápať ako futuritu, teda asynchrónne spracúvanú jednotku práce, ktorá nám niekedy v budúcnosti poskytne výsledok svojej činnosti. (Hoci spravidla nedokážeme exaktne predikovať okamih doručenia výsledku činnosti asynchrónnej úlohy, máme garanciu, že tento výsledok vždy získame.)

Proces stavby asynchrónnej metódy sa v programovacom jazyku C# 5.0 riadi týmito regulami:

1. Asynchrónna metóda je pomenovaná so sufixom „Async". Napríklad, ak sa pôvodná synchrónna metóda volala **NacitatData**, nová asynchrónna metóda sa bude volať **NacitatDataAsync**. Odporúčame vývojárom, aby túto pomenovaciu konvenciu dodržiavali za každých okolností. Klienti (alebo aj iní programátori) budú totiž ihneď vedieť, že pracujú s asynchrónnou metódou.

2. V hlavičke asynchrónnej metódy sa objavuje modifikátor **async**. Tento modifikátor je umiestnený pred identifikátorom asynchrónnej metódy a za špecifikátorom jej prístupových práv. Prítomnosť modifikátora **async** v hlavičke metódy je pre prekladač jazyka C# 5.0 jasným znamením o tom, že v tele tejto metódy budú vyznačené viaceré body prerušenia s prepojením na adekvátne asynchrónne úlohy.

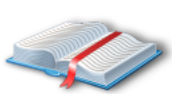
Dôležité: Prítomnosť modifikátora **async** v hlavičke metódy nespôsobuje jej automatickú transformáciu zo synchrónnej metódy na asynchrónnu metódu. Modifikátor **async** len naznačuje, že daná metóda sa správa asynchrónne. Vzorec správania asynchrónnej metódy je však determinovaný algoritmom, ktorý je naprogramovaný v tele tejto metódy. Nutnou podmienkou je, aby tento algoritmus pracoval asynchrónne.

3. Asynchrónna metóda môže byť bezparametrická alebo parametrická s vhodne zvolenou súpravou formálnych parametrov.

Dôležité: Parametrická asynchrónna metóda môže združovať hodnotové formálne parametre, no už nie odkazové (**ref**) a výstupné (**out**) formálne parametre. Povedané inými slovami, parametrická asynchrónna metóda získa vždy duplikáty pôvodných argumentov, ktoré boli špecifikované pri jej aktivácii.

Dátovým typom návratovej hodnoty asynchrónnej metódy je obvykle trieda **Task**, a to buď v konkrétnom (**Task**), alebo generickom (**Task<TResult>**) vyhotovení. Keď sa v tele asynchrónnej metódy objavuje príkaz **return v;**, kde **v** je výraz produkujúci objekt generickej triedy **Task<TResult>** s výsledkom asynchrónnej úlohy, volíme triedu **Task<TResult>** ako typ návratovej hodnoty tejto metódy. Na druhej strane, ak asynchrónna metóda neprodukuje príkazom **return** návratovú hodnotu, alebo príkaz **return** vôbec neobsahuje, vyberáme pre typ návratovej hodnoty tejto metódy konkrétnu triedu **Task**.

4. V tele asynchrónnej metódy sú prostredníctvom operátora **await** determinované body prerušenia aktuálneho toku spracovania tejto metódy. Ako sme spomenuli, počítame s tým, že asynchrónna metóda bude vykonávať jednu alebo aj viacero asynchrónnych úloh. Každá z asynchrónnych úloh je v našich úvahách futuritou. Vieme, že výsledok činnosti futurity príde v budúcnosti, zatiaľ (teda na začiatku spracovania asynchrónnej metódy) máme len prísľub, že tento výsledok nakoniec získame. Sémanticky je preto operátor **await** aplikovaný na futuritu, technicky na objekt triedy **Task**, respektíve **Task<TResult>**, do ktorého bude neskôr zaliaty výsledok činnosti futurity.



Poznámka: Prekladač jazyka C# 5.0 automaticky očakáva aspoň jeden výskyt operátora **await** v tele asynchrónnej metódy. Ak ho nenájde, vygeneruje varovanie. To znamená, že neprítomnosť operátora **await** v tele asynchrónnej metódy nevyústi do generovania chyby v čase prekladu. Avšak, bez

operátora **await** je táto metóda iba pseudo-asynchrónna, pretože jej algoritmus je vykonaný synchrónne s nepriaznivým efektom blokovania klienta.

5. V súvislosti s operátorom **await** je nutné pamätať na skutočnosť, že po jeho aplikácii sa riadenie okamžite vracia späť klientovi, a teda nespôsobuje jeho blokovanie. (Spomenuté blokovanie vznikalo nevyhnutne pri synchronizme a bolo považované za úzke hrdlo programu.)

2.4 Spracovanie asynchrónnych metód

Predpokladajme definíciu asynchrónnej metódy **NacitatDataAsync**:

```
private async Task<string> NacitatDataAsync(string cesta)
{
    // Algoritmus asynchrónneho načítania dát zo súboru.
}
```

Metóda je schopná asynchrónne vykonať proces načítania dát z externého textového súboru. V hlavičke metódy je zasadený modifikátor **async**, typom návratovej hodnoty metódy je **Task<string>** a metóda je parametrická s jedným formálnym parametrom reťazcového typu. Vieme, že asynchrónny algoritmus dekomponuje problém „Načítať textové dáta asynchrónne“ na problém „Spravovať množinu asynchrónnych úloh, ktoré zabezpečia asynchrónne načítanie dát“.

Na úrovni zvolenej abstrakcie nás v tejto chvíli neinteresuje technicky presný postup, ako budú pracovať jednotlivé asynchrónne úlohy tohto algoritmu. Naopak, veríme, že algoritmus „vie“, ako predostretý problém vyriešiť. My trváme na tom, aby algoritmus a, samozrejme, aj príslušná metóda, realizovali svoju prácu asynchrónne, a teda bez blokovania klientskej strany.

Pozrime sa teraz na stranu klienta. Klientom je spracovateľ udalosti konkrétneho objektu v programe s vizuálnym rozhraním.

Syntaktický skelet klienta je nasledujúci:

```
private async void Spracovatel_Udalosti(object sender, EventArgs e)
{
    // Prvá množina programových akcií.
    var ulohaAsync = NacitatDataAsync(cesta);
    // Druhá množina programových akcií.
    // Uskladnenie načítaných dát.
    string data = await ulohaAsync;
    // Finálne spracovanie textových dát.
}
```

Technicky ide o parametrického asynchrónneho spracovateľa udalosti bez návratovej hodnoty. Náš klient vykonáva dve množiny programových akcií a medzi nimi volá asynchrónnu metódu na načítanie dát z textového súboru. Po zavolaní je asynchrónnej metóde **NacitatDataAsync** odovzdaný argument, ktorým je absolútna cesta k fyzickému súboru s textovými dátami. Ako je zvykom, metóda poskytnutý argument uloží do svojho formálneho parametra. Očakávame, že pri najbližšej možnej príležitosti bude riadenie z asynchrónnej metódy **NacitatDataAsync** prenesené späť do spracovateľa udalosti. Akcia zmeny kontextu súvisí so spracovaním výrazu s operátorom **await** v tele asynchrónnej metódy **NacitatDataAsync**.

Asynchrónna metóda **NacitatDataAsync** poskytne klientovi okrem vrátenia kontroly aj objekt triedy **Task<string>**. Tento objekt reprezentuje futuritu, ktorá bude nakoniec uchovávať výsledok asynchrónneho spracovania metódy **NacitatDataAsync**, a teda konkrétnu textovú reprezentáciu dát z fyzického súboru. Keďže predpokladáme, že dáta ešte neboli asynchrónne načítané, môžeme na vrátenú futuritu nahliadať ako na prázdnu. Adjektívom „prázdna" poukazujeme na skutočnosť, že zatiaľ máme len prísľub na budúce doručenie konkrétnych textových dát. Medzitým, ako je asynchrónna metóda **NacitatDataAsync** zaneprázdnená načítavaním textových dát, môže pokračovať

spracovanie kódu na strane klienta. Z tohto titulu je možné, aby spracovateľ udalosti realizoval druhú množinu programových akcií, ktoré sú naprogramované v jeho tele. Pochopiteľne, predostretý výpočtový model očakáva, že žiadna z uskutočňovaných programových akcií druhej množiny nie je dátovo závislá od spracovania asynchrónnej metódy **NacitatDataAsync**.

Nakoniec prichádza na rad príkaz, v ktorom klient aplikuje operátor **await** na asynchrónne realizovanú úlohu (futuritu). Už sme vysvetlili, že jednou z praktických implikácií použitia operátora **await** je čakanie na dokončenie asynchrónne vykonávaného výpočtového procesu.

Pri úvahe o ďalšej interakcii medzi klientom (spracovateľom udalosti) a asynchrónnou metódou **NacitatDataAsync**, dospejeme k záveru, že sa môžu odohrať dva scenáre:

1. **Prvý scenár: Asynchrónna metóda stihne načítať textové dáta skôr, ako bude spracovaný výraz s operátorom await na strane klienta.** Pritom platí nasledujúce: pri skončení svojej činnosti vracia asynchrónna metóda „plnú" futuritu. Adjektívum „plná" používame na označenie takej futurity, ktorá už obsahuje konkrétne výstupné dáta. Skutočnosť, že asynchrónna operácia načítania textových dát je hotová, je pre klienta istotne potešujúca. A to najmä kvôli tomu, že prichystané textové dáta sa dajú ihneď vybrať z plnej futurity a použiť pri realizácii nasledujúcich programových operácií klienta.

2. **Druhý scenár: Asynchrónna metóda nestihne načítať dáta skôr, než dôjde k spracovaniu výrazu s operátorom await na strane klienta.** Vzhľadom na to, že klient prostredníctvom operátora **await** naznačil, že potrebuje požadované textové dáta (napríklad na realizáciu nastávajúcej programovej operácie), musí na ich dodanie čakať. Celkový čas čakania môže byť variabilne dlhý. Ak by mal spracovateľ udalosti nejakého klienta, odovzdalo by sa riadenie bez akéhokoľvek oneskorenia jemu. Tento klient by potom mohol pokračovať vo svojej pracovnej činnosti. No ak vychádzame z premisy, že náš spracovateľ udalosti žiadneho

priameho klienta nemá, nezostáva mu nič iné, iba čakať na poskytnutie plnej futurity asynchrónnou metódou.

2.5 Praktické riešenie: Asynchrónna analýza webovej stránky

Praktické riešenie, ktoré vyvinieme v programovacom jazyku C# 5.0, umožní používateľovi asynchrónne analyzovať žiadanú webovú stránku. Asynchrónny program pôsobí ako štandardná konzolová aplikácia systému Windows s nasledujúcim syntaktickým obrazom:

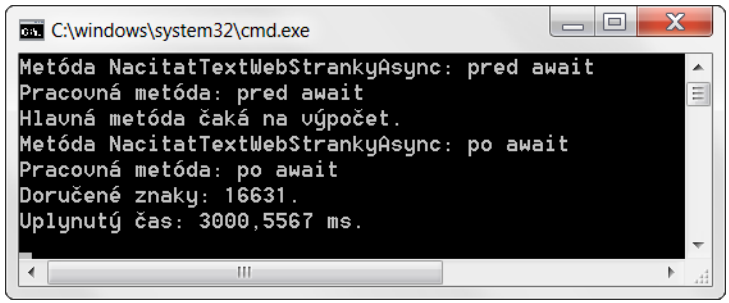
```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
using System.Net.Http;
using System.Text;
using System.Threading.Tasks;

namespace cs5_async
{
    class Program
    {
        static void Main(string[] args)
        {
            PracovnaMetoda();
            Console.WriteLine("Hlavná metóda čaká na výpočet.");
            Console.ReadLine();
        }
        // Definícia asynchrónnej pracovnej metódy.
        private static async void PracovnaMetoda()
        {
            Stopwatch sw = new Stopwatch();
            sw.Start();
            Task<string> ulohaNacitatStranku =
                NacitatTextWebStrankyAsync("http://www.fhi.sk");
            Console.WriteLine("Pracovná metóda: pred await");
            string textWebStranky = await ulohaNacitatStranku;
            Console.WriteLine("Pracovná metóda: po await");
            Console.WriteLine("Doručené znaky: {0}.",
```

```
                textWebStranky.Length.ToString());
            sw.Stop();
            Console.WriteLine("Uplynutý čas: {0} ms.",
                sw.Elapsed.TotalMilliseconds);
        }
        // Definícia asynchrónnej metódy na analýzu webovej stránky.
        private static async Task<string> NacitatTextWebStrankyAsync
        (string adresa)
        {
            Task<string> uloha = (new HttpClient()).GetStringAsync(adresa);
            Console.WriteLine("Metóda NacitatTextWebStrankyAsync: pred await");
            string textWebStranky = await uloha;
            Console.WriteLine("Metóda NacitatTextWebStrankyAsync: po await");
            return textWebStranky;
        }
    }
}
```

Komentár k zdrojovému kódu jazyka C# 5.0: Jadrom funkcionality programu je asynchrónna pracovná metóda (s rovnomenným názvom), ktorá generuje jednu asynchrónnu úlohu – tou je načítanie cieľovej webovej stránky a zistenie počtu textových znakov, ktoré táto webová stránka obsahuje. Textovú reprezentáciu webovej stránky získame vo forme textového reťazca, ktorý bude zapuzdrený do objektu triedy **Task<string>**. Pracovná metóda deleguje akciu asynchrónneho načítania textu webovej stránky na samostatnú (a rovnako asynchrónnu) metódu **NacitatTextWebStrankyAsync**. Táto metóda je parametrická, pričom na vstupe prijíma URL adresu požadovanej webovej stránky. Technicky k URI zdroju pristupujeme cez objekt triedy **HttpClient**, v súvislosti s ktorým aktivujeme parametrickú asynchrónnu metódu **GetStringAsync**. Tok programu je riadený bodmi prerušenia, ktoré sú vyznačené operátorom **await**. Operátor **await** je vždy aplikovaný na futuritu, teda asynchrónne realizovanú úlohu. Rovnako platí, že po aplikácii operátora **await** je riadenie odovzdané klientovi aktuálne volanej metódy.

Programová ukážka meria čas, ktorý je nutný na alokáciu operácie asynchrónneho načítania a analýzy cieľovej webovej stránky. Na výstupe program uvádza jednak počet znakov webovej stránky a rovnako aj čas na spracovanie zadaných asynchrónnych úloh (obr. 2.3).



Obr. 2.3: Výstup asynchrónneho praktického riešenia

Kapitola 3

Inovácie v jazyku C++

(podľa ISO-štandardu C++11)

Visual Studio

3 Inovácie v jazyku C++ (podľa ISO-štandardu C++11)

3.1 Lambda-výrazy

Lambda-výrazy predstavujú jednu z najvýznamnejších noviniek štandardu jazyka C++ vo verzii C++11. V prostredí jazyka C++ je lambda-výraz syntaktickou konštrukciou, vďaka ktorej môžeme elegantne definovať anonymnú funkciu (ktorú zvykneme často nazývať aj lambda-funkciou). Lambda-funkcia je robustná, pretože účinne kombinuje schopnosti funkčných objektov a smerníkov na funkcie. Funktor je objekt, ktorý disponuje svojím stavom a v každom platnom programovom kontexte naň môže byť aplikovaný operátor volania funkcie (**operator()**). Na druhej strane, manipuláciou so smerníkom na funkciu dokážeme pracovať s cieľovou funkciou nepriamo a napríklad vieme zabezpečiť jej paralelné spracovanie na samostatnom pracovnom vlákne. Lambda-výrazmi teda definujeme (spravidla na lokálnej úrovni) anonymné (parametrické či bezparametrické) lambda-funkcie, do tiel ktorých implementujeme požadovanú výpočtovú logiku.

Rozličné formy gramatiky lambda-výrazov si predvedieme na nasledujúcich praktických riešeniach.

3.1.1 Praktické riešenie: Práca s parametrickým lambda-výrazom v jazyku C++

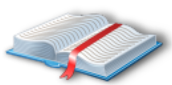
Parametrický lambda-výraz definujeme v jazyku C++ týmto spôsobom:

```
#include <iostream>
using namespace std;

int main()
{
```

```
    // Definícia lambda-výrazu.
    auto lambda = [](int a, int b) { return a + b; };
    // Zavolanie lambda-funkcie.
    cout << lambda(10, 4) << endl;
    return 0;
}
```

Komentár k zdrojovému kódu jazyka C++: Lambda-výraz je definovaný v prvom príkaze v tele hlavnej funkcie. Ako vidíme, ide o priraďovací príkaz, pričom definícia lambda-výrazu formuje inicializačný výraz tohto priraďovacieho príkazu. Lambda-operátor ([]) determinuje začiatočnú časť lambda-výrazu a predchádza jednak signatúru a rovnako aj telo lambda-výrazu.



Poznámka: Gramatika jazyka C++ predpisuje, že syntakticky vyzerá lambda-operátor ako preťažený operátor indexovania. To znamená novú sémantiku operátora [] pre vybraté programové kontexty, v ktorých dochádza ku generovaniu nových lambda-výrazov. Zadanie lambda-operátora je vždy povinné, pretože práve podľa jeho prítomnosti dokáže prekladač pri vykonaní syntaktickej analýzy spoznať začiatok nového lambda-výrazu.

Za lambda-operátorom stojí signatúra lambda-výrazu. Keďže sme povedali, že lambda-výraz je implicitnou anonymnou lambda-funkciou, vieme dodať, že tak lambda-funkcia ako aj lambda-výraz obsahujú svoju signatúru. Podľa mohutnosti množiny formálnych parametrov rozoznávame bezparametrické a parametrické lambda-výrazy (a, samozrejme, aj lambda-funkcie). V našom prípade je zhotovený lambda-výraz parametrický, pretože v jeho signatúre sa objavujú dva celočíselné formálne parametre.

Tip: Pri definícii bezparametrického lambda-výrazu sú zátvorky za lambda-operátorom prázdne. Gramatika jazyka C++ a rovnako aj jeho prekladač umožňujú úplné vynechanie signatúry pri tvorbe bezparametrických lambda-výrazov. Napríklad, obe nasledujúce definičné príkazy sú v jazyku C++ správne:

```
// Korektné definície bezparametrických lambda-výrazov.
auto lambda1 = []() { return 7; };
auto lambda2 = [] { return 7; };
```

> Hoci obe syntaktické formy sú korektné, nazdávame sa, že pre rýchlejšiu analýzu zdrojového kódu je lepšie vždy vizuálne uvádzať signatúru, a to aj v prípade, keď je prázdna.

Absolútna početnosť formálnych parametrov v signatúre lambda-výrazu nie je obmedzená, no lambda-výraz nemôže obsahovať variabilný počet formálnych parametrov (ako je známe, v jazyku C++ sa variabilita formálnych parametrov syntakticky definuje operátorom výpustky). Podobne, v signatúre lambda-výrazu sa nesmú vyskytovať anonymné formálne parametre a implicitné formálne parametre. Vyžaduje sa, aby každý formálny parameter uvádzal nielen svoj dátový typ, ale aj identifikátor.

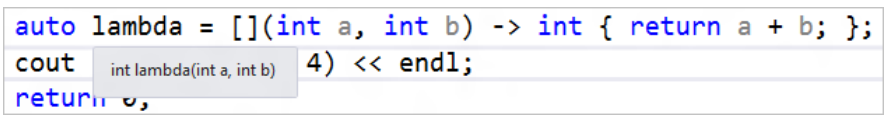
Telo lambda-výrazu formujú príkazy situované v bloku ohraničenom zloženými zátvorkami. V prípade predostretého lambda-výrazu je v tele prítomný len príkaz **return**, produkujúci návratovú hodnotu (ako súčet hodnôt formálnych parametrov). Typ návratovej hodnoty lambda-výrazu je prekladačom implicitne inferovaný podľa typov formálnych parametrov. Takmer vo všetkých prípadoch sa programátori môžu výhradne spoľahnúť na typovú dedukciu prekladača, ktorá je exaktná a funguje bezchybne. Za istých okolností, obvykle vtedy, keď existuje viacero variantov determinácie kandidátov na implicitne odvodený dátový typ návratovej hodnoty, je nutné, aby programátori prebrali zodpovednosť za určenie finálneho dátového typu návratovej hodnoty lambda-funkcie do vlastných rúk. Syntakticky tak robíme prostredníctvom preťaženého operátora ->, ktorý dodávame medzi signatúru a telo lambda-výrazu. Napríklad, skôr definovaný lambda-výraz (s implicitne inferovaným typom návratovej hodnoty) by vyzeral takto:

```
// Explicitne stanovený typ návratovej hodnoty lambda-výrazu.
auto lambda = [](int a, int b) -> int { return a + b; };
```

V jazyku C++ je definícia lambda-výrazu tým, čím je deklarácia lambda-výrazu v matematickom aparáte lambda-počtu. Keď prekladač jazyka C++ deteguje lambda-výraz, vykoná viacero závislých akcií. Najskôr deklaruje funktorovú triedu, ktorú vybaví požadovanou funkcionalitou. Potom túto triedu inštanciuje, na čo získa funktor, ktorému priradí logiku lambda-funkcie implementovanej lambda-výrazom. Povedané inými slovami, „back-end" pre lambda-výrazy tvoria funktorové triedy a ich objekty.

Pre potreby tejto publikácie je zmysluplné prijať abstrakciu, v ktorej pripustíme, že definovaný lambda-výraz smie byť priamo priradený do cieľovej lokálnej premennej s implicitne inferovaným dátovým typom (s využitím kľúčového slova **auto**). Naše úvahy môžeme v tomto smere rozvinúť aj ďalej: do lokálnej premennej ukladáme odkaz na lambda-funkciu, ktorá je zavedená príslušným lambda-výrazom.

O tejto skutočnosti nás presvedčí technológia IntelliSense vo chvíli, keď v editore zdrojového kódu presunieme kurzor myši na identifikátor lokálnej premennej uchovávajúcej definíciu lambda-výrazu (obr. 3.1).



Obr. 3.1: Dátový typ lokálnej premennej s definíciou lambda-výrazu

Zavolanie lambda-funkcie sa uskutočňuje spracovaním lambda-výrazu. Spracovanie lambda-výrazu je buď okamžité, alebo odložené. V praktickom riešení sme využili odložené spracovanie lambda-výrazu:

```
// Zavolanie lambda-funkcie.
cout << lambda(10, 4) << endl;
```

Pri odloženom spracovaní lambda-výrazu je asociovaná lambda-funkcia zavolaná neskôr než v momente definície lambda-výrazu. Vzhľadom na to, že lambda-funkcia je prístupná len cez lokálnu premennú s priradeným lambda-výrazom, musíme operátor volania funkcie aplikovať práve na túto premennú. Prirodzene, ak je volaná lambda-funkcia parametrická (čo je aj náš prípad), poskytujeme jej argumenty v adekvátnom počte a type.

Výraz **lambda(10, 4)** vykladáme ako odložené spracovanie lambda-výrazu, respektíve ako volanie lambda-funkcie. Vykonanie lambda-funkcie je totožné s realizáciou akejkoľvek inej platnej funkcie v jazyku C++.

Okamžité spracovanie lambda-výrazu a súčasne aj okamžité zavolanie lambda-funkcie sa odohráva ihneď po definovaní lambda-výrazu, a to v tomto syntaktickom znení:

```
int main()
{
    // Okamžité zavolanie lambda-funkcie.
    auto lambda = [](int a, int b) -> int { return a + b; }(8, 2);
    cout << lambda << endl;
    return 0;
}
```

Všimnime si, že operátor volania funkcie aplikujeme (spoločne s argumentmi) ihneď za telo lambda-výrazu. Táto syntaktická zmena kompletne mení sémantiku celého priraďovacieho príkazu. Po prvé, inicializačný výraz agreguje definíciu a spracovanie lambda-výrazu (alebo, z pohľadu lambda-počtu, spája deklaráciu a aplikáciu lambda-funkcie). Po druhé, vytvorený lambda-výraz je vzápätí spracovaný. To znamená, že dochádza k zavolaniu lambda-funkcie a vykonaniu príkazu, ktorý je umiestnený v jej tele. Tento príkaz produkuje návratovú hodnotu lambda-funkcie, a tá sa stáva zároveň hodnotou inicializačného výrazu. Táto hodnota (čiže súčet hodnôt formálnych parametrov lambda-funkcie) je v ďalšej etape priradená do cieľovej lokálnej premennej. Pochopiteľne, mení sa aj implicitne inferovaný dátový typ tejto premennej. Typom premennej už nie je lambda-funkcia, ale návratová hodnota lambda-funkcie. Keďže typom návratovej hodnoty lambda-funkcie je **int**, je tento typ aj typom lokálnej premennej. Podobne, ako v predchádzajúcom prípade, aj teraz môžeme overiť typovú charakteristiku lokálnej premennej prostredníctvom technológie IntelliSense.

3.1.2 Praktické riešenie: Komparácia práce lambda-výrazu a ekvivalentného funktora

V predchádzajúcom praktickom riešení sme vysvetlili proces definície a spracovania parametrického lambda-výrazu. Pred tým, ako boli do jazyka C++ zapracované lambda-výrazy, mohli sme identickú funkcionalitu na objektovej úrovni abstrakcie implementovať

pomocou funktorov. Pozrime sa teda, ako vyzerá deklarácia funktorovej triedy s rovnakým modelom použitia:

```
// Deklarácia funktorovej triedy.
class Funktor
{
private:
    int hodnota;
public:
    // Definícia bezparametrického konštruktora.
    Funktor() : hodnota(0) {}
    // Definícia parametrickej operátorovej metódy.
    int operator()(int a, int b)
    {
        hodnota = a + b;
        return hodnota;
    }
    // Definícia bezparametrickej prístupovej metódy.
    int get_hodnota() { return hodnota; }
};
```

Komentár k zdrojovému kódu jazyka C++: V programovaní charakterizujeme funktorovú triedu ako triedu so schopnosťou generovať objekty, na ktoré možno aplikovať operátor volania funkcie (**operator()**). Aby sme dosiahli tento cieľ, musíme deklarovať triedu s preťaženým operátorom volania funkcie. Hoci existuje viacero alternatív, ako preťažiť operátor volania funkcie, v predstavenom praktickom riešení sme operátorovú metódu definovali ako členskú funkciu funktorovej triedy. Je zrejmé, že operátorová metóda je parametrická a disponuje celočíselnou návratovou hodnotou. Logika zaliata v tele operátorovej metódy verne reflektuje prácu lambda-výrazu uvedeného v prvom praktickom riešení. Novinkou je archivácia vypočítanej hodnoty do súkromného dátového člena funktorovej triedy. Vďaka tejto kompozícii si funkčné objekty budú pamätať naposledy vypočítané hodnoty (k nim sa klienti týchto objektov dopracujú pomocou navrhnutej prístupovej metódy).

Deklarácia funktorovej triedy je len prvým krokom v celom procese. V ďalšej etape musíme túto triedu podrobiť inštanciácii:

```
int main()
{
    // Generovanie funktora.
    Funktor funktor1;
    // Simulácia práce lambda-výrazu pomocou funktora.
    int hodnota = funktor1(4, 6);
    cout << funktor1.get_hodnota() << endl;
    return 0;
}
```

Po automatickej inštanciácii získavame automatický funktor a v súvislosti s ním v druhom príkaze aplikujeme operátorovú metódu so vstupnými argumentmi. Operátorová metóda vykoná svoju prácu, pričom nám poskytne svoju návratovú hodnotu. Táto je následne priradená do lokálnej premennej (no pamätá si ju aj sám funktor).

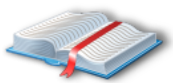
Funktorová trieda môže vystupovať aj ako šablónová funktorová trieda:

```
// Deklarácia šablónovej funktorovej triedy.
template<typename T> class Funktor
{
private:
    T hodnota;
public:
    Funktor() : hodnota(0) {}
    // Definícia generickej operátorovej metódy.
    T operator()(T a, T b)
    {
        hodnota = a + b;
        return hodnota;
    }
    T get_hodnota() { return hodnota; }
};

int main()
{
    // Automatická inštanciácia parametrizovaného funktora.
    Funktor<double> funktor1;
    // Simulácia práce lambda-výrazu pomocou parametrizovaného funktora.
    double hodnota = funktor1(4, 6);
```

```
    cout << funktor1.get_hodnota() << endl;
    return 0;
}
```

Komentár k zdrojovému kódu jazyka C++: Typový formálny parameter (**T**) sa objavuje na viacerých miestach v deklarácii šablónovej funktorovej triedy, avšak nás najviac zaujíma jeho pôsobenie v definícii generickej operátorovej metódy. Tu typový formálny parameter vystupuje v úlohe typu návratovej hodnoty operátorovej metódy a rovnako aj v úlohe typov oboch formálnych parametrov tejto metódy.



Poznámka: Keďže v tele operátorovej metódy je spracúvaný inicializačný výraz, ktorý spočítava hodnoty formálnych parametrov, bude tento výraz vyhodnotiteľný len vtedy, keď je prekladaču známa definícia binárneho operátora + pre dodané operandy. Za bežných okolností je všetko v poriadku, no pri aplikácii tohto operátora na dosiaľ nezavedené typy operandov môže dôjsť ku generovaniu chybovej výnimky. Riešením je preťažiť operátor + aj pre nové (a zrejme programátorom deklarované) dátové typy.

Inštanciáciou získavame parametrizovaný automatický funktor so špecializáciou pre reálny dátový typ s dvojnásobnou presnosťou. Na rozdiel od minulej funktorovej triedy, aktuálna funktorová trieda je generická. To znamená, že prekladač postupuje v dvoch krokoch: Najskôr z generického vzoru vytvorí konkrétnu (niekedy tiež vravíme finálnu) deklaráciu pôvodne generickej triedy. (Konkretizácia generickej predlohy sa dá uskutočniť len vtedy, keď prekladač pozná typové argumenty, čiže konkrétne typy, ktoré nahrádzajú generické typy.) Potom konkrétnu triedu inštanciuje a s vygenerovaným objektom manipuluje ako s ktorýmkoľvek objektom negenerickej triedy.

3.1.3 Praktické riešenie: Priradenie lambda-výrazu do systémového funktora

Lambda-výrazy a funktory dokážu spolupracovať. Hoci vývojári v jazyku C++ vedia sami naprogramovať také funktory, ktoré sú schopné prijať príslušné lambda-výrazy, v prostredí produktu Visual C++ 2012 je lepším riešením siahnuť po systémových

funktoroch. Systémové funktory sú objektmi šablónovej funktorovej triedy **function**, ktorá je deklarovaná v súbore functional. Nasledujúci program jazyka C++ demonštruje spoluprácu systémového funktora a lambda-výrazu. Zmyslom ukážky je navrhnúť algoritmus na precenenie produktov.

```
#include <iostream>
// Import hlavičkového súboru na podporu čŕt funkcionálneho programovania.
#include <functional>
using namespace std;


int main()
{
    // Tvorba poľa s cenami produktov.
    float produkty[] = {10.56f, 22.77f, 50.44f};
    const float pocetProduktov = 3;
    // Definícia lambda-výrazu.
    auto precenitProdukty = [&produkty](int i) -> void
    {
        produkty[i] *= 1.05;
    };
    // Inštanciácia a inicializácia systémového funktora.
    function<void (int)> funktor = precenitProdukty;
    for (int i = 0; i < pocetProduktov; i++)
    {
        // Precenenie kolekcie produktov použitím systémového funktora.
        funktor(i);
        cout << "Produkt " << i + 1 << " - nova cena: "
             << produkty[i] << endl;
    }
    return 0;
}
```

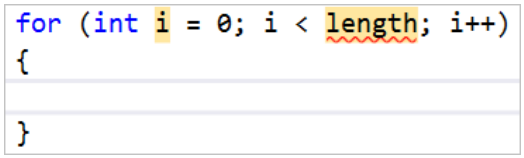
Komentár k zdrojovému kódu jazyka C++: Vzhľadom na to, že systémový funktor je objektom generickej funktorovej triedy, musíme do aktuálnej oblasti platnosti programu zaviesť hlavičkový súbor functional. Pre potreby praktického riešenia pracujeme s tromi produktmi, ktorých ceny sme zoskupili do poľa. Definovaný lambda-výraz je parametrický, pričom k externému poľu bude pristupovať odkazom.

Dôležité: Lambda-výrazy majú oprávnenie pristupovať k externým lokálnym objektom dvomi spôsobmi: hodnotou a odkazom. V našom prípade budeme k externému lokálnemu poľu pristupovať odkazom, pretože chceme meniť ceny produktov, čo je akcia, ktorá spôsobí modifikáciu aktuálneho stavu tohto poľa. Za týchto okolností používame preťažený referenčný operátor (&), ktorý aplikujeme na ten objekt, ku ktorému chceme pristupovať odkazom. Zatiaľ čo zápis **[&obj]** znamená prístup k lokálnej premennej **obj** odkazom, zápis **[=obj]** opisujeme ako prístup k danej premennej hodnotou. Pri prístupe hodnotou je lambda-výrazu odovzdaná kópia požadovanej lokálnej premennej. Prirodzene, lambda-výraz môže modifikovať prijatý duplikát premennej, no všetky vykonané zmeny sú trvácne len do opustenia tela lamda-výrazu. Pri konštrukcii lambda-výrazov vieme určiť implicitný spôsob zachytávania lokálnych entít, a to opäť pomocou operátorov = a &. Každý z operátorov je použitý v práve jednom vyhotovení na prvej pozícii v lambda-operátore, napríklad **[=]** alebo **[&]**. Ozrejmené pravidlá sa dajú výhodne kombinovať: trebárs zápis **[=, &mix]** znamená implicitné zachytávanie všetkých lokálnych objektov hodnotou, avšak okrem lokálneho objektu **mix**, ktorý bude získaný odkazom.

Interesantný je proces zrodu systémového funktora: trieda **function** je generická a jej špecializáciou je signatúra anonymnej lambda-funkcie, ktorá bola definovaná lambda-výrazom. Z kódu dedukujeme, že lambda-funkcia bude bez návratovej hodnoty, no parametrická s jedným celočíselným formálnym parametrom. Pri inicializácii je do systémového funktora dosadený odkaz na cieľovú lambda-funkciu. Aby všetko fungovalo správne, systémový funktor a spriaznená lambda-funkcia musia byť úplne konformní.

Precenenie kolekcie produktov sa odohráva v iteratívnej programovej konštrukcii prostredníctvom systémového funktora. Nové ceny produktov sú následne zobrazené na výstupe programu.

Tip: Editor zdrojového kódu produktu Visual C++ 2012 spolupracuje s expanzívnymi šablónami kódu. Tieto šablóny sú vopred pripravenými syntaktickými skeletmi, ktoré smú byť veľmi rýchlo vložené do editovaného zdrojového kódu a vzápätí programátorom upravené do finálnej podoby. Napríklad, po zadaní kľúčového slova **for** a stlačení tabulátora editor zdrojového kódu importuje expanzívnu šablónu pre cyklus **for** s hlavičkou a prázdnym telom (obr. 3.2).



Obr. 3.2: Expanzívna šablóna zdrojového kódu jazyka C++

Počnúc verziou 2012 produktu Visual C++ môžu vývojári zostrojovať svoje vlastné expanzívne šablóny zdrojového kódu. Prehľad dostupných expanzívnych šablón pre jazyk C++ získame po spustení nástroja **Code Snippets Manager** (ponuka **Tools** → položka **Code Snippets Manager**).

3.1.4 Praktické riešenie: Bezstavové lambda-výrazy

V programovaní chápeme bezstavové lambda-výrazy ako lambda-výrazy, ktoré nezachytávajú žiadne lokálne objekty. Takéto lambda-výrazy sú podľa smerníc štandardu C++11 implicitne pretypovateľné na ekvivalentné smerníky na funkcie. Túto funkcionalitu ukážeme v nasledujúcom praktickom riešení, v ktorom budeme pracovať s lambda-výrazom na výpočet Stirlingovho faktoriálu:

```
int main()
{
    // Definícia bezstavového lambda-výrazu.
    auto stirling = [](int n){ return sqrt(2 * 3.14f * n)
        * pow(n / 2.71f, n); };
    // Definícia a inicializácia ekvivalentného smerníka na funkciu.
    float (*pf)(int) = stirling;
    // Aplikácia lambda-výrazu.
    cout << stirling(5) << endl;
    // Aplikácia smerníka na funkciu.
    cout << pf(5) << endl;
    return 0;
}
```

Komentár k zdrojovému kódu jazyka C++: Do lokálnej premennej s identifikátorom **stirling** s implicitne inferovaným dátovým typom ukladáme parametrický lambda-výraz realizujúci výpočet aproximovanej hodnoty faktoriálu podľa Stirlinga. Pokračujeme definíciou smerníkovej premennej s názvom **pf**, ktorá dokáže uskladniť smerník na

ľubovoľnú cieľovú funkciu s návratovou hodnotou typu **float** a s jedným celočíselným formálnym parametrom.

Smerníkovú premennú zasadíme na ľavú stranu priraďovacieho príkazu, zatiaľ čo na pravú stranu tohto príkazu umiestnime lokálnu premennú **stirling**. Aby mohlo byť priradenie úspešne spracované, musí existovať správna verzia preťaženého priraďovacieho operátora. Ako „správnu" máme na mysli takú verziu operátora =, ktorá získa smerník na cieľovú anonymnú a parametrickú lambda-funkciu a priradí ho do príslušnej smerníkovej premennej. Syntakticky potom vieme vyčísliť približnú hodnotu pre 5! buď pomocou premennej **stirling**, alebo prostredníctvom smerníkovej premennej **pf**. Samozrejme, faktoriál bude vždy vypočítavať lambda-funkcia, a preto nás neprekvapí, že v oboch prípadoch dostaneme rovnaké výsledky.

3.1.5 Praktické riešenie: Vnorené lambda-výrazy

Keď do tela jedného lambda-výrazu vložíme iný lambda-výraz, tak navrhneme vnorený lambda-výraz. Pritom však chceme, aby medzi nadradeným a vnoreným lambda-výrazom existovala istá informačná väzba. Manipuláciu s vnoreným lambda-výrazom dokumentuje ďalší výpis zdrojového kódu jazyka C++:

```
int main()
{
    // Definícia nadradeného lambda-výrazu.
    auto lambda = [](int x)
    {
        // Definícia vnoreného lambda-výrazu.
        return [](int y) { return y * y; }(x * x);
    }(4);
    cout << lambda << endl;
    return 0;
}
```

Komentár k zdrojovému kódu jazyka C++: Nadradený lambda-výraz je parametrický, no nezachytáva žiadne externé objekty. Vnorený lambda-výraz je rovnako parametrický a tiež nepristupuje k žiadnym lokálnym objektom. Za zmienku stojí technika, ktorou sme

vytvorili želanú väzbu medzi nadradeným a vnoreným lambda-výrazom. Najlepšie túto techniku opíšeme tak, že budeme sledovať algoritmus spracovania nadradeného lambda-výrazu. Hoci nadradený lambda-výraz produkuje návratovú hodnotu, táto nebude stanovená skôr, než dôjde k spracovaniu vnoreného lambda-výrazu.

Syntaktický skelet definície vnoreného lambda-výrazu znie: **[](int y) { return y * y; }**. Logika začlenená do tela vnoreného lambda-výrazu je priamočiara: návratovou hodnotou tohto výrazu je druhá mocnina hodnoty jeho formálneho parametra. Po bližšom skúmaní však odhalíme, že definícia vnoreného lambda-výrazu je spojená s okamžitým spracovaním tohto výrazu, pričom vstupným argumentom pre vnorený výraz je druhá mocnina aktuálnej hodnoty formálneho parametra nadradeného lambda-výrazu. A práve na tomto mieste identifikujeme jedno z prepojení medzi nadradeným a vnoreným lambda-výrazom.

Možno nás napadne otázka, či vôbec môže byť v tomto kontexte použitý aplikačný výraz obsahujúci formálny parameter nadradeného lambda-výrazu. Odpoveď znie: áno, samozrejme, veď:

- Po prvé, formálny parameter nadradeného lambda-výrazu môže byť aplikovaný kdekoľvek v tele tohto lambda-výrazu (a vnorený lambda-výraz je predsa súčasťou tela nadradeného lambda-výrazu).

- Po druhé, existuje stopercentná garancia, že formálny parameter nadradeného lambda-výrazu bude vždy inicializovaný skôr, než bude použitý ako vstup pre vnorený lambda-výraz.

Kroky skúmaného algoritmu nás zatiaľ doviedli k okamžitému spracovaniu vnoreného lambda-výrazu. Ak zväčšíme rozlíšenie, ktorým nahliadame na zdrojový kód, postrehneme, že nadradený lambda-výraz je po svojej definícii tiež okamžite spracovaný. Vstupným argumentom je celočíselná konštanta 4.

Pokúsme sa zaznamenať pohyb tejto konštanty vo výpočtových operáciách:

1. Najskôr vystupuje konštanta 4 ako argument nadradeného lambda-výrazu.
2. Argumentom 4 je inicializovaný formálny parameter (**x**) nadradeného lambda-výrazu.
3. Formálny parameter nadradeného lambda-výrazu (**x**) je použitý vo vstupnom výraze (**x*x**) pri okamžitom spracovaní vnoreného lambda-výrazu. Teraz je zrejmé, že aplikácia vnorenej lambda-funkcie je realizovaná s argumentom 16.
4. Argumentom 16 je inicializovaný formálny parameter (**y**) vnoreného lambda-výrazu.
5. Vyhodnotí sa výraz **y*y** v tele vnoreného lambda-výrazu. Jeho hodnotou je 256 (16^2).
6. Vnorený lambda-výraz propaguje výslednú hodnotu 256 nadradenému lambda-výrazu.
7. Nadradený lambda-výraz túto hodnotu uloží do lokálnej premennej (**lambda**).

3.1.6 Praktické riešenie: Lambda-výrazy vyššieho rádu

Lambda-výraz vyššieho rádu je každý lambda-výraz, ktorý spĺňa aspoň jednu z týchto podmienok:

1. Vracia iný lambda-výraz vo forme svojej návratovej hodnoty.
2. Prijíma iný lambda-výraz vo forme vstupného argumentu.

Rovnaké pravidlá platia aj pre lambda-funkcie vyššieho rádu.

V nasledujúcom praktickom riešení predvádzame stavbu lambda-výrazu vyššieho rádu, ktorý je konformný s prvou z uvedených podmienok.

```
#include <iostream>
// Import potrebného hlavičkového súboru.
#include <functional>
```

```
using namespace std;

int main()
{
    // Definícia lambda-výrazu vyššieho rádu.
    auto lambda = [](int g) -> function<float (float)>
    {
        return [=](float h) -> float
        {
            return g / h;
        };
    };
    // Spracovanie lambda-výrazu vyššieho rádu.
    auto vysledok = lambda(7)(2);
    cout << vysledok << endl;
    return 0;
}
```

Komentár k zdrojovému kódu jazyka C++: V predloženom fragmente kódu diagnostikujeme definíciu lambda-výrazu vyššieho rádu. Tento lambda-výraz je parametrický (s celočíselným formálnym parametrom **g**) a s explicitne špecifikovaným dátovým typom svojej návratovej hodnoty. Pri determinácii typu návratovej hodnoty sme využili šablónovú funktorovú triedu **function** z hlavičkového súboru functional. Ako vidíme, lambda-výraz bude generovať nový lambda-výraz v podobe svojej návratovej hodnoty. Novo zostrojený lambda-výraz bude definovať parametrickú lambda-funkciu s návratovou hodnotou reálneho typu s jednoduchou presnosťou.

V tele lambda-výrazu je prítomný príkaz **return**, za ktorým je situovaná definícia nového lambda-výrazu, pôsobiaceho ako návratová hodnota nadradeného lambda-výrazu. Vyšetrením vráteného lambda-výrazu zistíme, že dokáže pristupovať hodnotou k lokálnym objektom (jediným platným lokálnym objektom je v tomto prípade formálny parameter hlavného lambda-výrazu). Ďalej analyzujeme parametrickú povahu tohto lambda-výrazu a telo s priamočiarym aritmetickým príkazom.

Spracovanie lambda-výrazu vyššieho rádu je odložené: uskutočňuje sa v momente, keď je spracovaný výraz **lambda(7)(2)**.

Pozrime sa bližšie na algoritmus spracovania tohto výrazu a rozoberme ho po jednotlivých krokoch:

1. **Spracovanie podvýrazu lambda(7).** Pri spracovaní tohto podvýrazu prekladač vie, že argumentom 7 bude inicializovaný formálny parameter (**g**) lambda-výrazu vyššieho rádu. Rovnako prekladač vie, že návratovou hodnotou tohto podvýrazu bude funktor, respektíve lambda-funkcia s takouto signatúrou: **function<float (float)>**. Zatiaľ má však prekladač málo údajov na to, aby vedel vygenerovať návratový lambda-výraz a podrobiť ho explicitnej exekúcii.

2. **Spracovanie výrazu lambda(7)(2).** Dodatočná špecifikácia argumentu pre formálny parameter vráteného lambda-výrazu umožní prekladaču inicializovať jeho formálny parameter a správne vyhodnotiť výraz v tele tohto lambda-výrazu. Na konci tohto reťazca akcií máme k dispozícii konkrétnu hodnotu (3.5f v našom prípade), ktorú prekladač ukladá do premennej **vysledok** s implicitne inferovaným dátovým typom.

Jazyk C++ nám, samozrejme, umožňuje spracovať lambda-výraz vyššieho rádu aj nasledujúcim spôsobom:

```
int main()
{
    auto lambda = [](int g) -> function<float (float)>
    {
        return [=](float h) -> float
        {
            return g / h;
        };
    };
    auto c = lambda(7);
    auto vysledok = c(2);
    cout << vysledok << endl;
    return 0;
}
```

Rozdielom je použitie samostatnej lokálnej premennej (**c**), ktorá slúži na uskladnenie návratového funktora (respektíve návratovej lambda-funkcie). Keď presunieme kurzor

myši nad identifikátor tejto premennej, uvidíme jej automaticky dedukovaný dátový typ: **std::function<float (float)>**. Na druhej strane, dátový typ lokálnej premennej **vysledok** nebude predstavovať funkčný objekt, ale konkrétnu reálnu hodnotu (typu **float**).

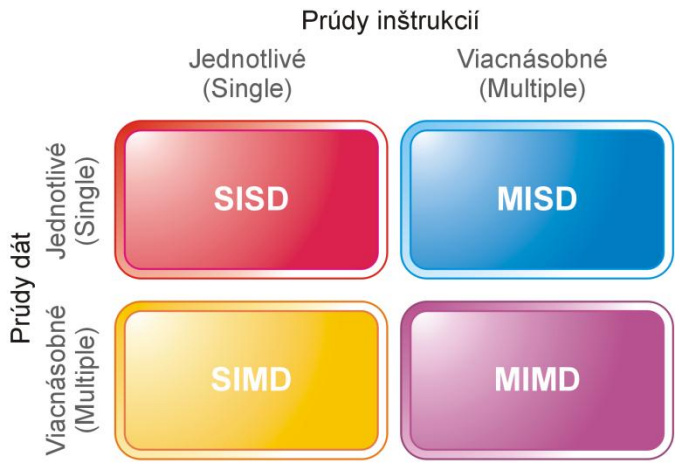
3.2 Automatický vektorizátor prekladača jazyka C++

Automatický vektorizátor je softvérový stroj, ktorý uskutočňuje samočinnú vektorizáciu pôvodného skalárneho zdrojového kódu. Vektorizátor je súčasťou prekladača jazyka C++ v implementácii pre vývojové prostredie Visual C++ 2012. Vektorizácia zdrojového kódu je sofistikovanou optimalizačnou technikou, ktorej hlavným cieľom je signifikantné zvýšenie výkonnosti spracovania operácií v iteratívnych programových konštrukciách (v jazyku C++ sa automatická vektorizácia týka predovšetkým cyklov **for**). Vektorizátor vykonáva transformáciu skalárneho kódu na vektorový, a to tak, že spracovanie existujúceho kódu akceleruje aplikovaním SIMD (angl. *Single-Instruction-Multiple-Data*) vektorových inštrukcií.

Základný pracovný model automatického vektorizátora je zachytený na obr. 3.3.

Obr. 3.3: Pracovný model automatického vektorizátora prekladača jazyka C++

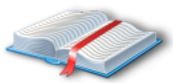
Flynnova taxonómia determinuje rôzne triedy mikroprocesorových architektúr, a to podľa ich schopnosti sekvenčného alebo paralelného spracovania inštrukčných prúdov v súvislosti s asociovanými dátovými prúdmi. Na obr. 3.4 je znázornená matica reprezentujúca Flynnovu taxonómiu.



Obr. 3.4: Maticová vizualizácia Flynnovej taxonómie

Ak je inštrukčná súprava mikroprocesora schopná využívať inštrukcie typu SIMD, znamená to, že dokáže v jednom hodinovom cykle vykonať jednu inštrukciu v súvislosti s viacerými dátovými elementmi. Štandardný skalárny kód využíva skalárne registre mikroprocesora. Pre skalárny register je typické, že dokáže uložiť práve jeden dátový element. Na druhej strane, vektorový kód spolupracuje s vektorovými registrami. Vektorový register je schopný uskladniť vektor dát, teda sériovú postupnosť dátových elementov. Kvantitatívnym atribútom každého registra (či už skalárneho, alebo vektorového) je jeho šírka. Prirodzene, šírka skalárneho registra je menšia ako šírka vektorového registra.

Predpokladajme, že zatiaľ čo skalárny register je široký 32 bitov, vektorový register je 4-násobne širší, pričom jeho absolútna šírka je 128 bitov. Uvažujme, že jeden dátový element predstavuje inštanciu typu s alokačnou kapacitou 32 bitov (teda 4 bajty). Potom je jasné, že vektorový register môže absorbovať 4 dátové elementy súčasne. Zmyslom vektorových inštrukcií triedy SIMD je konkurenčné spracovanie inštrukcie (respektíve inštrukčného prúdu) na všetkých dátových elementoch aktuálne alokovaných v príslušnom vektorovom registri. Model, ktorý sme predstavili, umožňuje dosiahnuť maximálne 4-násobný teoretický nárast výkonnosti programu.



Poznámka: Moderné mikroprocesory firiem Intel a AMD obsahujú hardvérovú implementáciu vektorových inštrukcií triedy SIMD. Spoločnosť Intel nazýva konkrétnu implementáciu týchto inštrukcií názvom SSE (angl. *Streaming SIMD Extensions*). Proces zavedenia spomínaných vektorových inštrukcií má viacgeneračnú genézu, a tak rozoznávame vektorové inštrukcie v generáciách SSE, SSE2, SSE3 a SSE4. Medzigeneračný progres spočíva v inkrementálnom dodávaní nových vektorových inštrukcií, ktoré rozširujú pôvodnú množinu týchto inštrukcií. Pre úplnosť výkladu dodajme, že firma AMD implementuje vektorové inštrukcie typu SIMD prostredníctvom technológie 3DNow!.

Prekladač jazyka C++, ktorý je súčasťou produktu Visual C++ 2012, vykonáva prostredníctvom svojho vektorizátora automatickú vektorizáciu skalárneho kódu. Uskutočnenie vektorovej transformácie je automatické vždy, keď je výstupom prekladača ostré zostavenie programu (v konfigurácii **Release**). V momente, keď zapneme voľbu na generovanie ostrého zostavenia, sa ihneď aktivujú tieto optimalizačné techniky:

- **Optimalizácia zameraná na dosiahnutie maximálnej možnej rýchlosti spracovania strojového kódu programu.** Tejto optimalizačnej technike prislúcha voľba **Optimization** aktivovaná hodnotou **Maximize Speed** (prepínač **/O2**).

- **Optimalizácia celého programu.** Tejto optimalizačnej technike zodpovedá voľba **Whole Program Optimization**, ktorá je prepnutá do stavu **Yes (/GL)**.

Pri generovaní cieľového programu v ladiacom režime (profil **Debug**) nie je automatická vektorizácia realizovaná. Tento štýl správania prekladača je pochopiteľný, pretože ladiace vyhotovenia programov nie sú nijakým spôsobom optimalizované.

3.2.1 Algoritmus konštrukcie cyklov na úspešnú implementáciu vektorovej transformácie

Automatický vektorizátor, ktorý implementuje prekladač prostredia Visual C++ 2012, sa koncentruje najmä na spracovanie samočinnej vektorizácie cyklov **for**. Preto sa zameriame na pracovný postup konštrukcie takých cyklov **for**, ktoré budú priateľské voči vektorizácii.

Ako vieme, každý cyklus **for** sa skladá z hlavičky a tela.

Pre hlavičku cyklu **for**, ktorý je navrhnutý s intenciou neskoršej vektorizácie, platia tieto zásady:

- Riadiaca premenná cyklu je celočíselná premenná dátového typu **int**.

- Riadiaca premenná cyklu je lokálna premenná vzhľadom na tento cyklus.

- Inicializačná hodnota, ktorá determinuje dolnú hranicu cyklu, môže vzísť z akéhokoľvek zmysluplného a v čase prekladu vyhodnotiteľného výrazu. Ak programátor použije inicializačný výraz, jeho hodnota musí byť celočíselného typu **int**, alebo musí byť implicitne konvertovateľná na celočíselnú hodnotu typu **int**. Samozrejme, inicializačnou hodnotou môže byť aj celočíselná konštanta.

- Hodnota, ktorá určuje hornú hranicu cyklu, musí byť konštantná počas celého spracovania cyklu. Existencia tejto nutnej podmienky je pochopiteľná, pretože cyklus pracuje najefektívnejšie vtedy, keď je jeho iteračný priestor známy čo možno najskôr (teda po vykonaní inicializačného a rozhodovacieho výrazu cyklu).

- Krok, akým cyklus postupuje po svojom iteračnom priestore, by mal byť inkrementačný s diferenciou 1. Tento stav vystihuje inkrementačný výraz **i++**, respektíve **++i**, ktorý je poslednou syntaktickou súčasťou hlavičky cyklu.

- Z pohľadu bezpečnosti je žiaduce, aby bola hodnota riadiacej premennej zvýšená priamo v modifikačnom výraze cyklu vždy po každej vykonanej iterácii a nikde inde. Poznamenajme, že narušenie tohto princípu prináša problémy aj pri konštrukcii bežných cyklov, bez požiadavky na ich vektorizáciu.

Na druhej strane, telo cyklu **for**, ktorý má podstúpiť úspešnú vektorizáciu, musí byť konformné s týmito predpismi:

- Algoritmus, ktorý je zaliaty do tela cyklu, musí byť rozumne zložitý. Vždy sa preto usilujeme o to, aby sme skonštruovali taký algoritmus, ktorý je dostatočne robustný na to, aby bol prekladačom vektorizovaný. V prípade, ak je algoritmus v tele cyklu triviálny, nebude skalárny cyklus transformovaný do svojej vektorovej povahy.

- Algoritmus implementovaný v tele cyklu je skomponovaný tak, aby vykonával operácie s prvkami polí a kolekcií. Tento algoritmus by nemal volať externé funkcie, realizovať viaccestné vetvenie toku programu, či explicitne manipulovať s chybovými výnimkami. Podmieňovací spôsob je v predchádzajúcom súvetí použitý zámerne. Prítomnosť spomenutých syntaktických konštrukcií nie je a priori zakázaná, no znižuje šancu na úspešnú (v zmysle maximálne výkonnú) vektorovú transformáciu cyklu. Hoci aktivačné výrazy volajúce externé funkcie môžu predstavovať potenciálne riziko, volanie „vektorových" verzií týchto funkcií je bezpečnou programátorskou technikou.

Tip: Viaceré matematické funkcie, ktoré sú súčasťou štandardnej knižnice jazyka C++ (a deklarované v súbore cmath), sú implementované v skalárnom aj vektorovom vyhotovení. Preto ich môže bez problémov použiť aj algoritmus tela cyklu, ktorý bude vektorizovaný.

- Keď sa v tele jedného cyklu vyskytuje iný cyklus, potom je možné vektorizovať len vnorený cyklus. Toto pravidlo má všeobecnú platnosť, odhliadnuc od skutočnej hĺbky zanorenia cyklu. Keď to bude možné, prekladač vektorizuje práve jeden, a síce najhlbšie vnorený cyklus.

- Zvýšenú pozornosť si vyžaduje analýza cyklov, v ktorých sa objavujú dátové väzby medzi jednotlivými iteráciami. Spravidla platí, že diagnostika takýchto dátových závislostí vylučuje správnu vektorizáciu cyklu. Dátové väzby vznikajú napríklad vtedy, keď výstup jednej iterácie cyklu pôsobí ako vstup pre inú iteráciu tohto cyklu. V naznačenej situácii je potrebné algoritmus tela cyklu pretransformovať tak, aby boli eliminované akékoľvek nepriaznivé dátové interferencie.

- Optimálne je, keď algoritmus operuje na celočíselných poliach typu **int**, respektíve na reálnych poliach typov **float** a **double**. Stranou by mali zostať znakové polia typu **char** a polia krátkych celých čísel typu **short** (v znamienkovej i neznamienkovej interpretácii).

- Redukčné operácie (ako je napríklad výpočet kumulatívneho súčtu hodnôt prvkov poľa) je nutné pre reálne polia (typov **float** a **double**) vykonávať so zapnutou voľbou prekladača **/fp:fast**.

- Optimalizátor prekladača nie je schopný automaticky vektorizovať cyklus vtedy, keď sú aktívne nasledujúce prepínače: **/kernel**, **/arch:IA32**, **/favor:ATOM**, **/O1** alebo **/Os**.

3.2.2 Praktický príklad automatickej vektorizácie cyklu

Pre potreby praktického odskúšania možností automatickej vektorizácie cyklu prekladačom založíme nový projekt štandardnej konzolovej aplikácie jazyka C++ a pridáme doň jeden zdrojový súbor (.cpp) s týmto syntaktickým obrazom:

```
#include <iostream>
using namespace std;

int main()
{
    const int rozsah = 1000;
    // Definície vektorových polí.
    int poleA[rozsah], poleB[rozsah], poleC[rozsah];
```

```
    // Cyklus inicializuje dve polia.
    for(int i = 0; i < rozsah; i++)
    {
        poleA[i] = poleB[i] = i;
    }
    // Cyklus inicializuje tretie pole.
    for(int i = 0; i < rozsah; i++)
    {
        poleC[i] = poleA[i] * poleB[i];
    }
}
```

Komentár k zdrojovému kódu jazyka C++: Logika predostretého programu je triviálna: generujeme 3 celočíselné polia, ktoré vzápätí inicializujeme. Vo výpise rozoznávame dva cykly **for**: prvý cyklus inicializuje polia **poleA** a **poleB**, zatiaľ čo druhý cyklus inicializuje tretie pole (**poleC**) pomocou predchádzajúcich polí. Rýchlou analýzou všetkých zúčastnených cyklov zistíme, že ich hlavičky aj telá spĺňajú všetky podmienky na úspešnú vektorizáciu.

Na zistenie skutočnosti, ktoré cykly programu boli vektorizované, musíme preložiť program v ostrom zostavení (**Release**) s aktívnym prepínačom **/Qvec-report:2**.

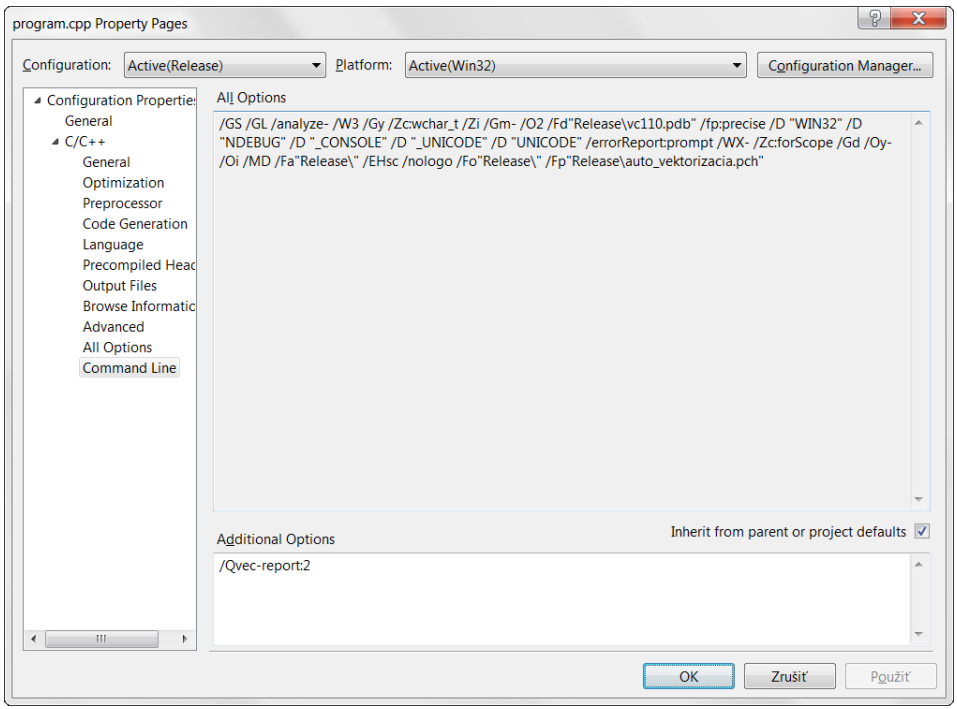
Prekladač jazyka C++ je citlivý na dva diagnostické prepínače, ktoré sa týkajú automatickej vektorizácie:

1. prepínač **/Qvec-report:1**: výstupom tohto prepínača sú cykly, ktoré boli úspešne vektorizované.
2. prepínač **/Qvec-report:2**: výstupom tohto prepínača sú úspešne aj neúspešne vektorizované cykly, pričom pre naposledy uvedené cykly sú uvedené aj príčiny zlyhania vektorovej transformácie.

V rámci tejto knihy budeme vždy aplikovať prepínač **/Qvec-report:2**. Jeho nastavenie pre príslušný zdrojový súbor jazyka C++ uskutočníme takto:

1. V podokne **Solution Explorer** klepneme na zdrojový súbor programu pravým tlačidlom myši a z lokálnej ponuky vyberieme príkaz **Properties**.

2. V dialógovom okne **Property Pages** vyhľadáme uzol **C/C++** → **Command Line**.
3. Do textového poľa **Additional Options** zadáme textovú charakteristiku prepínača, teda **/Qvec-report:2** (obr. 3.5).

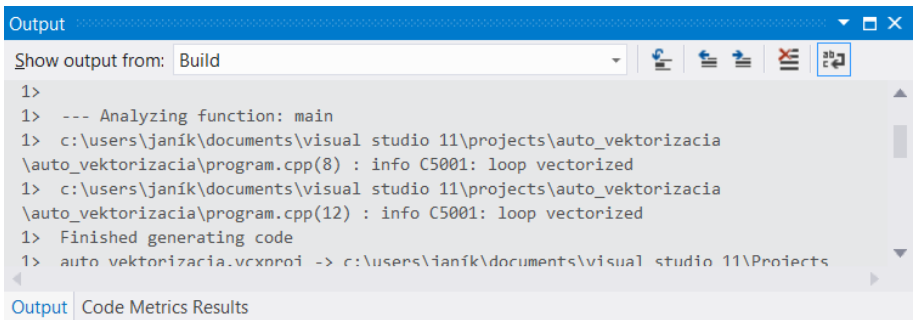


Obr. 3.5: Nastavenie diagnostického prepínača na dokumentáciu procesu automatickej vektorizácie zdrojového kódu

4. Vykonanú zmenu potvrdíme aktiváciou tlačidla **OK**.

Napokon uskutočníme zostavenie projektu jazyka C++ v ostrom vyhotovení.

Po zostavení projektu preskúmame informačný obsah podokna **Output** (obr. 3.6).



Obr. 3.6: Dokumentácia úspešnosti procesu vektorizácie po zostavení programu

Ako vidíme, oba cykly (nachádzajúce sa na riadkoch 8 a 12) boli vektorizované, čo dosvedčuje správa „*info C5001: loop vectorized*".

3.2.3 Základný pracovný algoritmus automatického vektorizátora

Automatický vektorizátor pracuje podľa tohto základného algoritmu:

1. Vektorizátor vyhľadá cykly **for** vo všetkých prekladových jednotkách programu jazyka C++.

2. Vektorizátor analyzuje cykly v záujme zistenia, či ich možno podrobiť vektorovej transformácii. Počas tejto prvotnej analytickej etapy vektorizátor najskôr vyšetruje, či je predmetný cyklus vhodným kandidátom na vektorovú transformáciu. Pozornosť vektorizátora sa sústreďuje na detekciu dátových väzieb medzi jednotlivými iteráciami cyklu. Vektorizátor si rovnako všíma aj prítomnosť už vysvetlených problematických techník a programových konštrukcií, ktoré by mohli pôsobiť proti vektorizácii.

3. Vektorizátor realizuje bezpečnostnú analýzu, počas ktorej zisťuje, či sa dá vektorizácia implementovať korektne. Korektnosť v tomto kontexte znamená, že správanie cyklu po vektorovej transformácii bude identické so správaním

pôvodného skalárneho cyklu. Ešte presnejšie, vektorový cyklus bude produkovať rovnaké výstupy ako ekvivalentný skalárny cyklus (samozrejme, pri totožnej množine vstupných dát).

4. Vektorizátor spracuje výkonnostnú analýzu podmnožiny cyklov, ktoré boli v predchádzajúcom kroku tohto algoritmu označené ako bezpečné na vektorizáciu. Zmyslom výkonnostnej analýzy je kvantifikovať pozitívny nárast rýchlosti spracovania zväzkov iterácií cyklu po jeho vektorizácii. Rýchlosť spracovania vektorovej verzie cyklu sa vždy porovnáva s rýchlosťou spracovania skalárnej verzie tohto cyklu. Dodajme, že vektorová transformácia bude uplatnená len pri tých cykloch, ktoré sú bezpečné a ktorým vektorizácia prinesie merateľný výkonnostný efekt.

5. Výsledkom analýzy vektorizátora je zoznam cyklov **for**, ktoré sú vhodnými kandidátmi na vektorizáciu. Do týchto cyklov budú následne implementované vektorové inštrukcie.

3.2.4 Praktická výkonnostná analýza automatickej vektorizácie

Výkonnosť automatickej vektorizácie budeme skúmať na programe jazyka C++, ktorý inicializuje polia. Kompletný syntaktický obraz programu je nasledujúci:

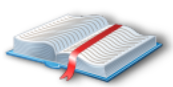
```
#include <iostream>
#include <Windows.h>

using namespace std;

int main()
{
    const int rozsah = 100;
    // Definície vektorových polí.
    int poleA[rozsah], poleB[rozsah], poleC[rozsah];
    // Inicializácia polí poleA a poleB.
    for(int i = 0; i < rozsah; i++)
    {
```

```
        poleA[i] = poleB[i] = i;
    }
    time_t t0, tN;
    t0 = GetTickCount64();
    for(int p = 0; p < 100000; p++)
    {
        // Inicializácia poľa poleC.
        for(int i = 0; i < rozsah; i++)
        {
            poleC[i] = poleA[i] * poleB[i];
        }
    }
    tN = GetTickCount64();
    cout << "Cas spracovania: " << tN - t0 << " ms." << endl;
    return 0;
}
```

Komentár k zdrojovému kódu programu jazyka C++: V programe vytvárame dovedna 3 vektorové polia, ktoré postupne inicializujeme. Predmetom nášho ďalšieho šetrenia bude inicializácia tretieho poľa (s identifikátorom **poleC**). Aby sme zvýšili zaťaženie systému, inicializáciu tohto poľa spracúvame 100000-krát. Celkový čas exekúcie inicializácie poľa **poleC** zaznamenávame s presnosťou na milisekundy (ms).



Poznámka: Programové stopky simuluje použitá funkcia **GetTickCount64**, ktorej deklaráciu nájdeme v hlavičkovom súbore Windows.h. Skutočná presnosť funkcie **GetTickCount64** je poznačená chybou 10 – 16 ms. To je však pre potreby nášho praktického riešenia akceptovateľná odchýlka, a preto ju pripúšťame. Na druhej strane, ak by sme chceli pracovať priamo s čítačmi a nie s milisekundami, museli by sme použiť časovač so schopnosťou vysokého rozlíšenia.

V tab. 3.1 uvádzame kvantifikovanú výkonnosť programu v troch zostavovacích režimoch:

1. **Ladiaci režim zostavenia (Debug)** – nie sú aplikované žiadne optimalizácie prekladača.
2. **Ostrý režim zostavenia (Release)** – sú aplikované všetky optimalizácie prekladača, vrátane automatickej vektorizácie.

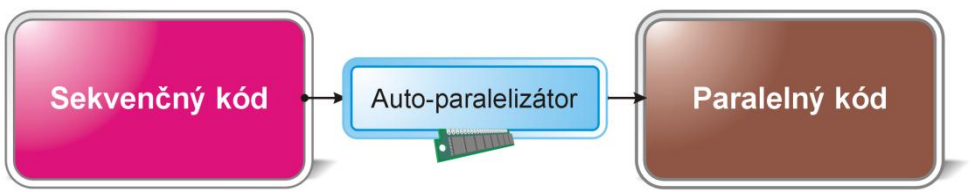
Tab. 3.1: Nameraná výkonnosť programu s implementáciou automatickej vektorizácie

Celkový čas spracovania programu v ladiacom režime (bez automatickej vektorizácie)		**Celkový čas spracovania programu v ostrom režime (s automatickou vektorizáciou)**	
Meranie	Celkový čas (ms)	Meranie	Celkový čas (ms)
1.	140	1.	31
2.	172	2.	32
3.	141	3.	46
4.	124	4.	31
5.	125	5.	31

Priemerný čas spracovania inicializačného procesu je v ladiacom režime 140,2 ms. Naopak, priemerný čas uskutočnenia inicializačného procesu je v ostrom režime 31,2 ms. Na základe nameraných údajov môžeme konštatovať, že optimalizácie spoločne s automatickou vektorizáciou priniesli 4,1-násobný nárast výkonu.

3.3 Automatický paralelizátor prekladača jazyka C++

Automatický paralelizátor je súčasťou prekladača jazyka C++, ktorý je implementovaný v produkte Visual C++ 2012. Úlohou automatického paralelizátora je vyhľadať sekvenčne spracúvané iteratívne konštrukcie a previesť ich do paralelnej formy (obr. 3.7).



Obr. 3.7: Pracovný model automatického paralelizátora prekladača jazyka C++

Paralelné cykly, ktoré vzniknú v procese automatickej paralelizácie z pôvodných sekvenčných cyklov, majú tieto vlastnosti:

- **Paralelný cyklus zavádza techniku dátového paralelizmu.** Ako vieme, dátový paralelizmus využíva dátovú fragmentáciu inštancie použitej dátovej štruktúry (ktorou je najčastejšie pole objektov alebo kolekcia objektov) s tým, že v súvislosti s viacerými dátovými fragmentmi je vykonávaný jeden paralelný algoritmus. Dátový paralelizmus je typický fragmentáciou iteračného priestoru paralelného cyklu. Výsledkom tohto fragmentačného procesu je existencia navzájom nezávislých (často vravíme tiež diskrétnych) zväzkov iterácií cyklu, ktoré je bezpečné a výhodné realizovať súbežne.

- **Paralelný cyklus dokáže využiť všetku disponibilnú výpočtovú kapacitu viacjadrových mikroprocesorov a viacprocesorových strojov.** Maximalizáciu výkonnosti spracovania paralelného cyklu na viacerých hardvérových vláknach viacjadrových mikroprocesorov chápeme ako hlavný cieľ paralelizmu. Zatiaľ čo iterácie sekvenčného cyklu sú vykonávané v sériovej postupnosti na jednom (obvykle primárnom) vlákne programu, paralelné zväzky iterácií paralelného cyklu sú uskutočňované súčasne na viacerých (spravidla pracovných) vláknach programu.

- **Výkonnosť paralelného cyklu je vždy vyššia ako výkonnosť sekvenčného cyklu.** Hoci na exaktné stanovenie výkonnosti paralelného cyklu existujú sofistikované matematické vzťahy, na rýchly test stačí, keď zmeriame celkové časy spracovania sekvenčného a paralelného cyklu a vypočítame ich pomer, teda:

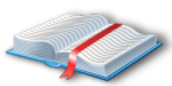
$$k_{NV} = \frac{T_{SC}}{T_{PC}}$$

kde:

  - k_{NV} je koeficient nárastu výkonnosti paralelného cyklu.
  - T_{SC} je celkový čas spracovania sekvenčného cyklu.
  - T_{PC} je celkový čas spracovania paralelného cyklu.

Napríklad, ak sekvenčný cyklus vykoná svoju prácu za 20 sekúnd a paralelný cyklus za 5 sekúnd, je zrejmé, že koeficient nárastu výkonnosti paralelne realizovaného cyklu je 4.

Samozrejme, uvedená premisa platí iba pre korektne transformované paralelné cykly. V tomto smere predpokladáme, že implementácia dátového paralelizmu do sekvenčného cyklu spôsobí (v bežných podmienkach) takmer lineárny nárast jeho výkonnosti (vo vzťahu k úplne vyťaženým výpočtovým uzlom počítačového systému).



Poznámka: Paralelné cykly, ktoré paralelizátor automaticky skonštruuje zo sekvenčných cyklov, musia vždy preukazovať vyššiu výkonnosť ako pôvodné sekvenčné cykly. Podmienka strojového zvýšenia výkonnosti cyklu jeho paralelizáciou je preto považovaná za nutnú. Ak by výsledkom automatickej paralelizácie nebol kvantifikovateľný nárast výkonnosti cyklu, prekladač by vygeneroval informačné hlásenie a ponechal by cyklus v sekvenčnej podobe. Porovnajme tento prístup s explicitnou implementáciou dátového paralelizmu programátorom. Tu sa totiž môže stať, že neodborne zavedený paralelizmus vygeneruje v konečnom dôsledku zníženie výkonnosti paralelného cyklu. Za týchto okolností bude paralelný cyklus pracovať menej efektívne ako ekvivalentný sekvenčný cyklus. Príčiny vedúce k tomuto stavu sú rôzne: prehliadnutím latentných dátových väzieb medzi iteráciami cyklu počínajúc a zanesením chyby spočívajúcej v súperení programových vlákien o zdieľaný dátových zdroj končiac.

- **Paralelný cyklus vznikne zo sekvenčného cyklu len vtedy, keď je to bezpečné a efektívne.** Z pohľadu analýzy považuje automatický paralelizátor podmienky bezpečnosti a efektívnosti za nutné, čo znamená, že musia byť, v záujme úspešnej paralelnej transformácie sekvenčného cyklu, vždy bezpodmienečne splnené. Keď paralelizátor analyzuje sekvenčný cyklus, zisťuje, či jeho premena na adekvátny paralelný cyklus bude bezpečná. Bezpečnosť v tomto kontexte chápeme ako zachovanie správnosti toku a determinizmu práce algoritmu, ktorý je zavedený v tele sekvenčného cyklu. Povedané inými slovami, paralelný cyklus musí produkovať vždy rovnaké výstupy ako sekvenčný cyklus (samozrejme, pri totožných vstupoch). Deterministické správanie paralelného cyklu definujeme ako

jeho schopnosť generovať v rôznych časoch rovnaké výstupy na základe identických vstupných dát. Menej formálne, ak by paralelný cyklus v desiatich reláciách svojho spracovania poskytoval 9-krát správny výsledok a 1-krát nesprávny výsledok, jeho správanie by nebolo deterministické.

Podmienka efektívnosti je priamo spojená s výkonnosťou paralelného cyklu. Podotknime, že analýzu efektívnosti transformácie sekvenčného cyklu na paralelný cyklus vykonáva automatický paralelizátor až po úspešnej bezpečnostnej analýze. (Ak by paralelná transformácia nebola bezpečná, práca paralelizátora končí, pričom analyzovaný cyklus bude ponechaný v originálnom sekvenčnom vyhotovení.) Výsledkom analýzy efektívnosti je kvalifikované rozhodnutie paralelizátora o tom, či je racionálne previesť sekvenčný cyklus na jeho paralelný ekvivalent. Kladná odpoveď reflektuje skutočnosť, že paralelný cyklus bude vždy preukazovať vyššiu výkonnosť ako pôvodný sekvenčný cyklus.

V teoretickej informatike existujú tri základné stupne nárastu výkonnosti paralelných cyklov voči sekvenčným cyklom:

1. **Sublineárny nárast výkonnosti paralelného cyklu.** Sublineárny (teda menej ako lineárny) nárast výkonnosti paralelného cyklu charakterizujeme vtedy, keď pri spracovaní tohto cyklu na *n*-procesorovom počítači je generovaný výkonnostný nárast menší ako *n*.

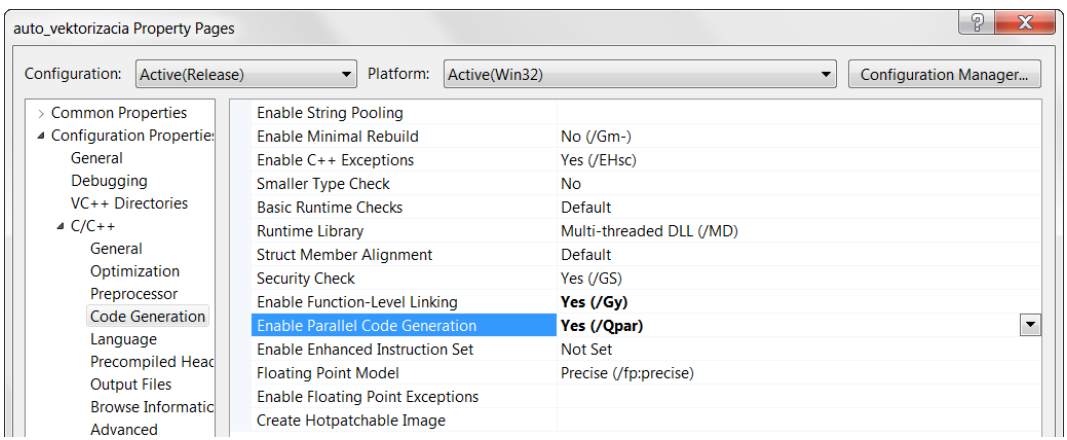
2. **Lineárny nárast výkonnosti paralelného cyklu.** O lineárnom náraste výkonnosti paralelného cyklu hovoríme vtedy, keď pri realizácii tohto cyklu na *n*-procesorovom počítači je produkovaný výkonnostný nárast rovnajúci sa *n*.

3. **Superlineárny nárast výkonnosti paralelného cyklu.** Superlineárny (teda viac ako lineárny) nárast výkonnosti paralelného cyklu diagnostikujeme vtedy, keď pri spracovaní tohto cyklu na *n*-procesorovom počítači je zistený výkonnostný nárast väčší ako *n*.

Pri praktickom paralelnom programovaní sme obvykle spokojní vtedy, keď paralelný algoritmus zakomponovaný v tele paralelného cyklu generuje takmer lineárny nárast svojej výkonnosti.

3.3.1 Komparácia pracovných modelov automatického vektorizátora a automatického paralelizátora

Ako sme vysvetlili, automatický vektorizátor samočinne vektorizuje všetky cykly, ktoré sú priateľské voči vektorovej transformácii. Vektorizátor vykonáva svoju činnosť implicitne, vývojári nemusia aktivovať žiadne špeciálne prepínače či vykonávať iné zmeny vo vlastnostiach projektu a vývojového prostredia. Vektorizátor je povolaný do práce vždy, keď realizujeme ostré zostavenie programu jazyka C++. Na druhej strane, automatický paralelizátor nie je implicitne aktivovaný. Služby tohto optimalizačného stroja si preto musíme explicitne vyžiadať prepnutím voľby **Enable Parallel Code Generation** do pozície **Yes (/Qpar)**. Cesta, vedúca k tejto voľbe, vyskytujúcej sa vo vlastnostiach projektu programu jazyka C++, je nasledujúca: **Configuration Properties** → **C/C++** → **Code Generation** → **Enable Parallel Code Generation** (obr. 3.8).



Obr. 3.8: Zapnutie voľby na automatickú paralelizáciu zdrojového kódu

Pokúsme sa teraz porovnať pracovné modely vektorizátora a paralelizátora. Vektorizátor realizuje vektorizáciu cyklov **for**. Pri tomto procese vektorizátor využíva vektorové (XMM) registre mikroprocesora, do ktorých načíta a uskladní vektorové pole dátových objektov. Vektorové registre sa delia na celočíselné vektorové registre a reálne vektorové registre.

Ak predpokladáme, že šírka jedného celočíselného XMM-registra je 128 bitov (čiže 16 bajtov), potom možno do tohto registra súčasne uložiť 4 dátové objekty celočíselného typu **int**. Na všetky dátové objekty uskladnené vo vektorovom registri sa dá paralelne aplikovať jedna vektorová SIMD-mikroinštrukcia, ktorá zmení aktuálne stavy spracúvaných dátových objektov. Dobrou správou je, že operácia nanesenia jednej vektorovej SIMD-mikroinštrukcie na konečný počet dátových objektov sa uskutoční v jednom hodinovom cykle mikroprocesora.

Pre proces úspešnej vektorizácie sú významné nasledujúce skutočnosti:

1. **Existencia dátových vektorových registrov mikroprocesora v dostatočnom počte a v dostatočnej alokačnej kapacite.** Registre tvoria najrýchlejšiu vrstvu pamäťovej hierarchie mikroprocesora. Keďže ide o vysokorýchlostnú pamäť s prístupovým časom na úrovni 1 nanosekundy, je výroba tejto pamäte značne drahá. Tým je vysvetlený fakt, prečo je alokačná kapacita registrov spravidla veľmi malá, pohybujúca sa v jednotkách bajtov. Napriek tomu, na úspešnú realizáciu vektorových SIMD-mikroinštrukcií potrebujeme adekvátny počet vektorových registrov. Vždy sa pritom snažíme alokovať všetky registre, ktoré máme k dispozícii. V tomto smere platí priama úmera: čím väčší počet vektorových registrov mikroprocesor obsahuje, tým viac dátových objektov môžeme paralelne spracovať prostredníctvom jednej SIMD-mikroinštrukcie.

2. **Schopnosť mikroprocesora explicitne spracovať vektorové SIMD-mikroinštrukcie nad dátovými objektmi, ktoré sú situované vo vektorových registroch.** Všetky súčasné mikroprocesory spĺňajú túto podmienku. Napríklad, firma Intel po prvýkrát inštalovala podporu spracovania SIMD-mikroinštrukcií do procesora Intel Pentium s multimediálnymi rozšíreniami MMX. Mikroprocesory

architektúry Intel Core 2 a Intel Core i7 integrujú mechanizmy na spracovanie tokov SIMD-mikroinštrukcií v niekoľkých generáciách (aktuálnou generáciou bola v čase tvorby tohto diela generácia SSE 4.2).

3. **SIMD-mikroinštrukcie zavádzajú paralelizmus na najnižšej možnej, teda na hardvérovej úrovni.** Technicky presne označujeme tento druh paralelizmu ako paralelizmus na úrovni mikroinštrukcií (angl. *Instruction-Level-Paralelism*, ILP).

Správne implementovaná automatická vektorizácia môže znamenať zvýšenie výkonnosti programu aj pri jeho spracovaní na jednoprocesorovom počítačovom systéme. Stačí, aby bol osadený mikroprocesor schopný explicitnej exekúcie tokov SIMD-mikroinštrukcií.

Automatický paralelizátor transformuje sekvenčné cykly na ich paralelné ekvivalenty. Táto transformácia sa deje implementáciou dátového paralelizmu do sekvenčného cyklu. Jadrom implementácie dátového paralelizmu je fragmentácia iteračného priestoru cyklu. Výsledkom fragmentačného procesu sú diskrétne zväzky iterácií cyklu, ktoré možno realizovať súbežne na viacerých jadrách *n*-jadrových mikroprocesorov.

Pre proces úspešnej automatickej paralelizácie sú významné tieto skutočnosti:

1. **Diskrétna povaha iteratívnych algoritmov implementovaných v telách cyklov.** V tomto kontexte chápeme diskrétny algoritmus ako algoritmus, ktorý technikou problémovej dekompozície rozkladá riešený problém na jednotlivé podproblémy, medzi ktorými neexistujú žiadne priame funkčné a/alebo dátové väzby. V situácii, keď paralelizátor deteguje také závislosti medzi iteráciami cyklu, ktoré porušujú podmienku diskrétnosti, dôjde k zastaveniu jeho činnosti a predmetný cyklus nebude paralelizovaný (zostáva teda v pôvodnej sekvenčnej verzii).

2. **Dostatočne veľká množina vstupných dát cyklov.** Ak má byť sekvenčný cyklus implicitne prevedený na paralelný cyklus, potom je významná mohutnosť množiny vstupných dát, na ktorej tento cyklus vykonáva naprogramované

operácie. V záujme eliminácie výkonnostných penalizácií vznikajúcich z titulu dátovej lokality je vždy rozumné pracovať s lineárne usporiadanou (v zmysle pamäťovej alokácie) množinou vstupných dát. Pri voľbe stratégie maximálnej možnej výkonnostnej optimalizácie kódu je rovnako zmysluplné fragmentovať vstupné dáta tak, aby ich načítanie do vyrovnávacích pamätí mikroprocesora bolo čo najefektívnejšie.

3. **Dostatočné veľké pracovné zaťaženie cyklu.** Sekvenčný cyklus obsahuje sekvenčný iteratívny algoritmus. Keďže paralelizátor mení sekvenčný cyklus na paralelný cyklus, je pochopiteľné, že táto transformácia znamená aj premenu sekvenčného iteratívneho algoritmu na paralelný iteratívny algoritmus. Cyklus, ktorý má byť paralelizátorom transformovaný do paralelnej formy, má zmysel paralelizovať len vtedy, keď je implementovaný algoritmus v tele cyklu výpočtovo intenzívny. Podľa tohto predpisu sa riadi aj paralelizátor, a preto sekvenčné cykly s nízkym pracovným zaťažením bývajú vylúčené z procesu paralelizácie.

Prekladač jazyka C++ postupuje pri preklade zdrojového kódu v ostrom zostavovacom režime takto:

1. Prekladač prostredníctvom automatického vektorizátora prevedie všetky vhodné skalárne cykly na vektorové cykly.

2. Prekladač prostredníctvom automatického paralelizátora prevedie všetky vhodné sekvenčné cykly na paralelné cykly.

Ak bude práca vektorizátora a súčasne aj paralelizátora úspešná, získavame množinu paralelne vykonávaných vektorových cyklov. To je najlepší možný stav, ku ktorému sa dá dospieť.

3.3.2 Praktický príklad automatickej paralelizácie cyklu

Do zdrojového súboru nového projektu jazyka C++ umiestnime syntaktický skelet nasledujúceho programu:

```
#include <iostream>
#include <Windows.h>

using namespace std;
const int rozsah = 100000;
// Definície globálnych vektorových polí so stanovenými rozsahmi.
int poleA[rozsah], poleB[rozsah], poleC[rozsah];

int main()
{
    time_t cas0, casN;
    cas0 = GetTickCount64();
    // Prvý kandidát na automatickú paralelizáciu.
    #pragma loop(hint_parallel(0))
    for(int i = 0; i < rozsah; i++)
    {
        // Inicializácia dvoch vektorových polí.
        poleA[i] = 2 * i - 1;
        poleB[i] = 7 * i % 2;
    }
    // Druhý kandidát na automatickú vektorizáciu.
    #pragma loop(hint_parallel(0))
    for(int i = 0; i < rozsah; i++)
    {
        // Inicializácia tretieho poľa.
        poleC[i] = poleA[i] * poleB[i];
    }
    casN = GetTickCount64();
    cout << "Cas spracovania: " << casN - cas0 << " ms." << endl;
    return 0;
}
```

Komentár k zdrojovému kódu jazyka C++: Predmetom bližšej analýzy sa stanú dva inicializačné cykly **for**. Aby sme automatickému paralelizátoru naznačili, že si prajeme paralelizáciu týchto cyklov, uvádzame pred ich hlavičky špeciálnu pragmu **#pragma loop(...)**.

Všeobecný zápis tejto pragmy znie:

```
#pragma loop(hint_parallel(n))
```

kde:

- **n** je počet pracovných vlákien, ktoré budú súbežne spracúvať iteratívny algoritmus paralelného cyklu.

Keď chceme presne stanoviť početnosť pracovných vlákien, dosadíme za argument **n** konkrétnu kladnú celočíselnú konštantu, napríklad 4 pre štyri pracovné vlákna. Ak by sme však radi ponechali rozhodnutie o počte pracovných vlákien na paralelné behové prostredie jazyka C++, položíme **n** rovné nule. V takomto prípade vytvorí paralelné behové prostredie adekvátny počet pracovných vlákien.

Poznámka: Uvádzanie pragmy **#pragma loop(...)** nie je povinnou aktivitou v živote vývojára. Paralelizátor vyhľadáva vhodných kandidátov na paralelizáciu aj bez explicitného použitia tejto pragmy. No nazdávame sa, že jej injektovanie do zdrojového kódu je užitočnou technikou, ktorou sme schopní upozorniť paralelizátor na výpočtovo intenzívne cykly, ktoré by mali byť transformované do paralelnej podoby.

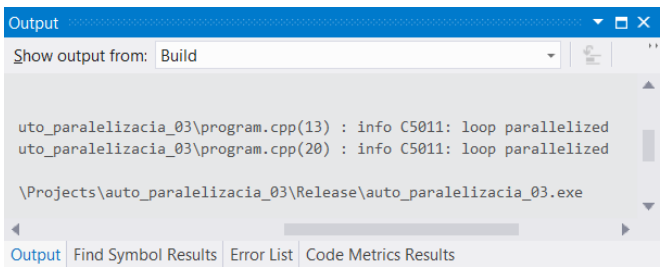
Pred ostrým zostavením programu uskutočníme tieto akcie:

1. **Zapneme podporu pre automatickú paralelizáciu prekladača jazyka C++.** To spravíme tak, že v dialógovom okne projektových vlastností zapneme voľbu **Enable Parallel Code Generation**.

2. **Aktivujeme diagnostický prepínač Qpar-report:2.** Tak prikážeme prekladaču podávať informačné správy o tom, ktoré cykly boli úspešne podrobené automatickej paralelnej transformácii a ktoré nie. Diagnostický prepínač **Qpar-report:2** vložíme do textového poľa **Additional Options**, ktoré nájdeme v sekcii **Command Line** vlastností projektu jazyka C++.

Tip: Podobne ako automatický vektorizátor, aj automatický paralelizátor exponuje dva diagnostické prepínače, a to **Qpar-report:1** a **Qpar-report:2**. Zatiaľ čo prvý prepínač generuje informačné správy len pre úspešne paralelizované cykly, druhý prepínač vypisuje informačné správy pre všetky (či už úspešne, alebo neúspešne) paralelizované cykly. Vývojári môžu voliť medzi obidvomi prepínačmi podľa aktuálnych požiadaviek vytváraného programu.

Vygenerujeme priamo spustiteľný program jazyka C++ v ostrom zostavovacom režime (**Release**). Z textovej dokumentácie podokna **Output** sa dozvieme, ako dopadol proces automatickej paralelizácie cyklov programu (obr. 3.9).

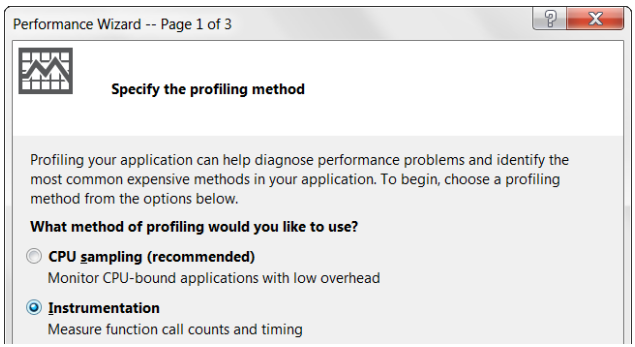


Obr. 3.9: Informačný výstup automatického paralelizátora

Ako vidíme, paralelizátor uskutočnil paralelnú transformáciu cyklov **for** vyskytujúcich sa na riadkoch 13 a 20 zdrojového súboru programu jazyka C++.

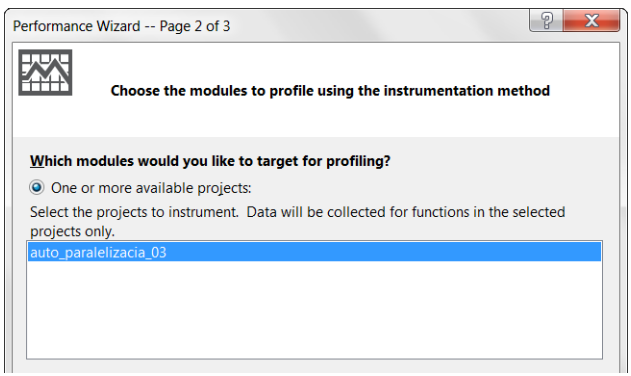
Na zistenie, v akej miere paralelný program využíva výpočtovú kapacitu počítačového systému, môžeme vyskúšať jeho profilovanie technikou inštrumentácie. Pritom postupujeme takto:

1. Otvoríme ponuku **Analyze** a klikneme na položku **Launch Performance Wizard**.
2. Spustí sa sprievodca **Performance Wizard**. V prvom kroku sprievodcu nastavíme ako profilovaciu metódu inštrumentáciu (voľba **Instrumentation**).



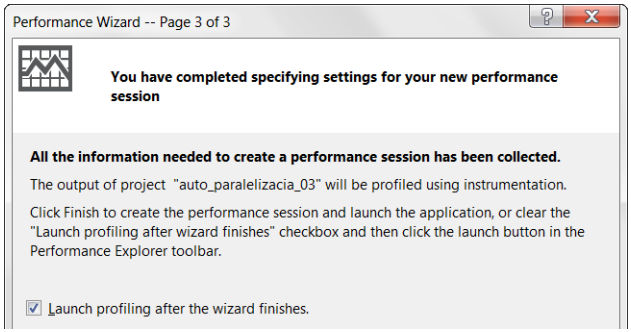
Obr. 3.10: Výber inštrumentácie ako metódy na profilovanie paralelného kódu

3. V druhom kroku sprievodcu potvrdíme profilovanie aktuálne otvoreného projektu (**One or more available projects**).



Obr. 3.11: Výber aktuálneho projektu ako cieľa profilovacieho procesu

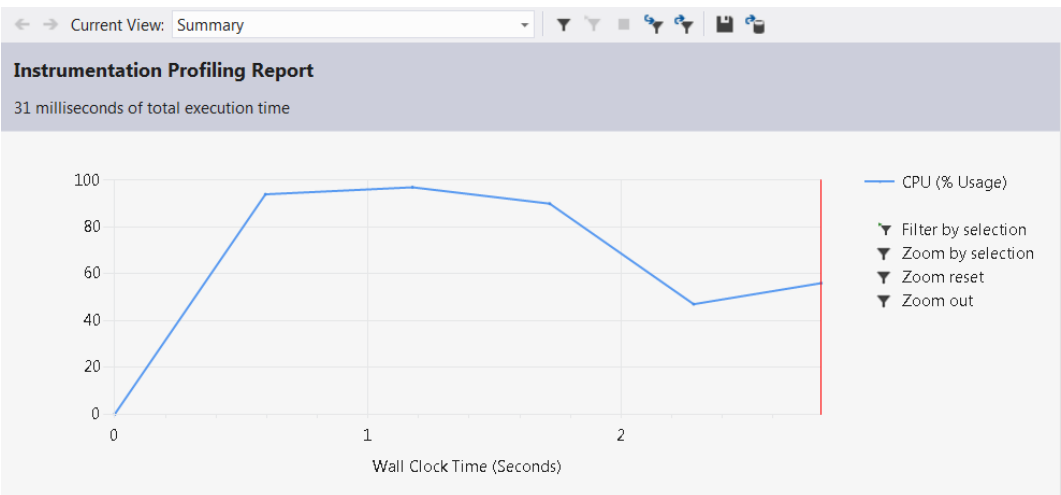
4. V posledkom kroku sprievodcu sa uistíme, že je aktívna voľba okamžitého spustenia profilovacieho procesu po ukončení sprievodcu (**Launch profiling after the wizard finishes**).



Obr. 3.12: Kontrola aktivácie voľby na okamžité spustenie profilovacieho procesu

5. Klepneme na tlačidlo **Finish**.

Profilovací program spustí náš paralelný program a vykoná skenovanie jeho výkonnosti počas celkového času spracovania tohto programu v operačnom systéme. Len čo sa profilovanie skončí, profilátor zobrazí výstupnú správu **Instrumentation Profiling Report** (obr. 3.13).



Obr. 3.13: Výstupná správa inštrumentačnej profilovacej techniky

Paralelný program dosiahol celkový čas spracovania v dĺžke 31 ms. Graf, ktorý vidíme vo výstupnej správe, reprezentuje relatívne využitie výpočtovej kapacity mikroprocesorov paralelným programom.

3.4 Jazyk C++ a vývoj WinRT-programov pre systém Windows 8

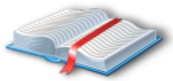
Jednou z najcennejších inovácií jazyka C++ je jeho schopnosť implementovať a generovať cieľové programy pre nové WinRT-rozhranie systému Windows 8. Nové programy, ktoré sú konformné s WinRT-rozhraním, nazývame termínom WinRT-programy.

WinRT-programy sú špecifické vo viacerých ohľadoch, a preto sa s nimi zoznámime bližšie:

- **Vizuálne rozhranie WinRT-programov je navrhnuté pomocou deklaratívneho jazyka XAML.** Vývojové prostredie Visual C++ 2012 obsahuje grafický editor, ktorý podporuje paradigmu vizuálneho programovania v jazyku XAML a súčasne ponúka robustné kolekcie užitočných ovládacích prvkov a komponentov ihneď použiteľných na praktické programovanie. Použitie sofistikovaného grafického editora zvyšuje rýchlosť zvládania všetkých pracovných procesov súvisiacich so stavbou programov kompatibilných s novým vizuálnym rozhraním systému Windows 8.

- **Logika WinRT-programov je tvorená v jazyku C++ s komponentovými rozšíreniami CX (angl. *Component Extensions*).** Vďaka tomu vznikol nový jazyk, ktorý nesie názov C++/CX. Jazyk C++/CX vychádza z gramatiky a špecifikácie jazyka C++ v súlade s najnovším ISO-štandardom (ISO/IEC 14882:2011), no pridáva početné syntakticko-sémantické rozšírenia, ktoré akcelerujú vývoj programov vo vzťahu k programovému rozhraniu WinRT. A hoci sa fragmenty zdrojového kódu jazyka C++/CX nezriedka ponášajú na syntaktické skelety jazyka

C++/CLI, je dôležité mať na pamäti skutočnosť, že výstupom prekladača jazyka C++/CX je rýdzo natívny kód.



Poznámka: Základy WinRT-rozhrania sú položené na technológii COM a natívnych rozhraniach COM-tried. Ako vývojári v jazyku C++ môžeme zhotovovať programy a komponenty kompatibilné s WinRT-rozhraním aj priamo v jazyku C++ v kooperácii s technológiou COM. Rýchlejším riešením je však priame upotrebenie jazyka C++/CX, ktorý automatizuje veľa dosiaľ programátormi manuálne realizovaných činností.

- **WinRT-programy môžu bežať na variabilných počítačových zariadeniach.** K podporovaným zariadeniam patria štandardné počítačové stanice, počítače s dotykovými obrazovkami (známe ako počítače typu All-In-One), notebooky, ultrabooky a tablety. Pritom očakávame, že na všetkých strojoch je nainštalovaný operačný systém Windows 8, a to buď v plnej, alebo odľahčenej (WinRT) verzii (ktorá je zameraná predovšetkým na tablety). Variabilita sa však netýka len typov počítačových zariadení, ale aj charakteru ich hardvérovej infraštruktúry. WinRT-programy dokážu bežať na systémoch s mikroprocesorovými architektúrami x86, x64 a ARM.

- **WinRT-programy majú špecifický životný cyklus.** Po aktivácii (spustení) poskytujú WinRT-programy používateľovi svoje služby. Používateľ sa môže prepínať z jedného WinRT-programu do iného WinRT-programu. Keď používateľ prenesie zameranie na iný WinRT-program, dosiaľ aktívny WinRT-program sa po istom čase dostane do stavu výkonovo efektívnej suspendácie. Novinkou je, že používateľ bežne neukončuje WinRT-programy explicitne. O ukončenie spracovania WinRT-programu sa stará systém Windows 8 automaticky. Technicky je správa WinRT-programov riadená algoritmami preemptívneho viacúlohového spracovania s pseudoparalelnými, respektíve rýdzo paralelnými exekučnými modelmi (v závislosti od miery vyspelosti hardvérovej infraštruktúry počítačového systému).

- **WinRT-programy využívajú paradigmu asynchrónneho programovania.** V skutočnosti každá operácia, ktorej celkový čas spracovania si vyžiada viac ako 50 milisekúnd, je naprogramovaná ako asynchrónna operácia. WinRT-programy rutinne realizujú asynchrónne operácie viažuce sa na vstupno-výstupné operácie, prácu s pamäťovými zdrojmi či manipuláciu so sieťovými dátovými prúdmi.

- **WinRT-programy sú inštalované na pás dlaždíc systému Windows 8.** Každý WinRT-program vlastní jednu dlaždicu, ktorá je po nasadení tohto programu samočinne umiestnená na pás dlaždíc štartovacej ponuky operačného systému. Dlaždica WinRT-programu môže pôsobiť ako aktívna – za týchto okolností má dynamický charakter a program je schopný prostredníctvom aktívnej dlaždice vysielať relevantné informácie používateľovi aj vtedy, keď s nim používateľ aktuálne nepracuje.

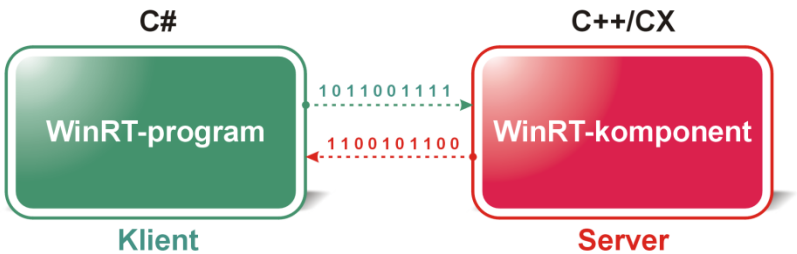
- **WinRT-programy sú distribuované elektronicky pomocou obchodu Windows.** Obchod Windows (angl. *Windows Store*) je digitálny megamarket spoločnosti Microsoft, ktorý je cieľovou stanicou všetkých WinRT-programov. Ešte predtým, ako sa WinRT-program dostane do vybratého katalógu obchodu Windows, musí prejsť schvaľovacím procesom. Po úspešnej certifikácii smie byť WinRT-program vystavený do katalógu obchodu Windows ako voľne dostupný alebo platený. Používatelia si WinRT-programy preberajú priamo z obchodu Windows. Prevzatie a inštalácia nového WinRT-programu sú akcie, ktorých uskutočnenie si vyžiada len niekoľko sekúnd systémového času.

3.5 Praktická ukážka vývoja a použitia WinRT-komponentu v jazyku C++/CX

V tomto návodnom postupe sa zoznámime s nasledujúcimi procesmi:

1. Proces vytvorenia nového WinRT-komponentu v jazyku C++/CX.

2. Proces použitia zhotoveného WinRT-komponentu v prostredí WinRT-programu, ktorý bol vytvorený v jazyku C#.



Obr. 3.14: Model interoperability
medzi WinRT-programom a WinRT-komponentom

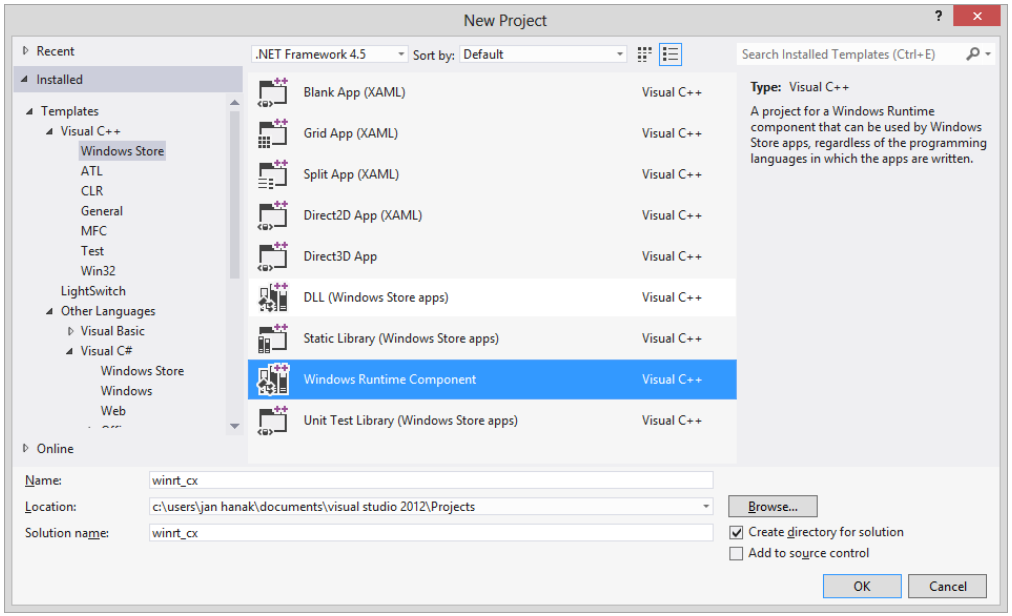
WinRT-komponent, ktorý vyvinieme, bude vypočítavať maximálny možný nárast paralelného programu voči sekvenčnému programu pomocou Amdahlovho zákona.

Dôležité: Úspešné dokončenie prezentovanej praktickej ukážky je založené na premise, že vývojár pracuje na počítači s nainštalovaným systémom Windows 8 a s prostredím Visual Studio 2012. WinRT-komponenty jazyka C++/CX nemožno vyvíjať na počítačovej stanici so systémom Windows 7.

3.5.1 Vytvorenie nového WinRT-komponentu v jazyku C++/CX

Projekt nového komponentu, ktorý je konformný s rozhraním WinRT, založíme v jazyku C++/CX takto:

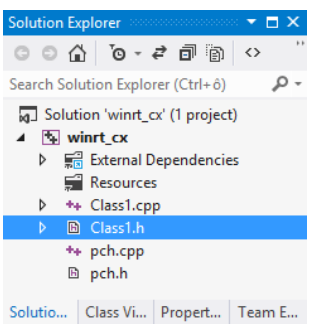
1. Na úvodnej stránke **Start Page** aktivujeme príkaz **New Project**.
2. V stromovej štruktúre vyberieme cestu **Templates** → **Visual C++** → **Windows Store** a špecifikujeme projektovú šablónu **Windows Runtime Component**.
3. Nový projekt pomenujeme ako winrt_cx (obr. 3.15).



Obr. 3.15: Založenie projektu nového WinRT-komponentu v jazyku C++/CX

4. Po klepnutí na tlačidlo **OK** sprievodca vygeneruje všetky projektové súbory.

V editore zdrojového kódu je automaticky otvorený hlavičkový súbor Class1.h. Pohľad do podokna **Solution Explorer** nám však prezradí, že okrem hlavičkového súboru bol samočinne prichystaný aj rovnomenný implementačný súbor (Class1.cpp). Rovnako sme schopní identifikovať prítomnosť predkompilovaných súborov pch.h a pch.cpp (obr. 3.16).



Obr. 3.16: Kolekcia projektových súborov nového WinRT-komponentu

Implicitne vygenerovaný zdrojový kód, ktorý je uložený v hlavičkovom súbore Class1.h, má takúto podobu:

```
#pragma once

namespace winrt_cx
{
    public ref class Class1 sealed
    {
    public:
        Class1();
    };
}
```

Pragma **#pragma once** nariaďuje práve jedno spracovanie editovaného hlavičkového súboru v procese prekladu zdrojového kódu programu jazyka C++/CX. Jadrom každého WinRT-komponentu je primárny menný priestor (**winrt_cx**) a primárna odkazová trieda (zatiaľ implicitne pomenovaná ako **Class1**). Životné cykly objektov odkazových (**ref**) tried jazyka C++/CX sú automaticky spravované mechanizmom počítania odkazov (respektíve referencií). Nutnou podmienkou primárnej odkazovej triedy WinRT-komponentu je jej zapečatený charakter, ktorý je syntakticky vyjadrený modifikátorom **sealed**. Dodajme, že zapečatená trieda nemôže vystupovať ako bázová trieda, z ktorej by boli v procese verejnej jednoduchej dedičnosti odvodzované nové podtriedy. Požiadavka na zapečatenosť primárnej odkazovej triedy je opodstatnená, pretože jazyk C++/CX v súčasnej implementácii explicitne nepodporuje princíp dedičnosti. V tele primárnej odkazovej triedy je situovaný verejný prototyp bezparametrického inštančného konštruktora.

Pokračujme inšpekciou zdrojového súboru Class1.cpp:

```
// Class1.cpp
#include "pch.h"
#include "Class1.h"

using namespace winrt_cx;
using namespace Platform;

Class1::Class1()
```

```
{
}
```

Direktívy predprocesora vkladajú obsahy determinovaných hlavičkových súborov. Deklarácie **using** zase realizujú zavedenie menných priestorov **winrt_cx** a **Platform**. Nakoniec prichádza definícia bezparametrického konštruktora implicitnej triedy.

Aby sme primárnu odkazovú triedu vybavili požadovanou funkcionalitou, a teda algoritmizáciou Amdahlovho zákona, uskutočníme tieto akcie:

1. Syntaktický skelet primárnej odkazovej triedy upravíme v hlavičkovom súbore Class1.h takto:

```
public ref class ParalelneVypocty sealed
{
private:
    float narastVykonnosti;
public:
    ParalelneVypocty();
    float VypocitatNV(float tS, float tP, int n);
};
```

   Do súkromnej sekcie triedy sme doplnili definíciu reálneho dátového člena, ktorý bude archivovať vypočítaný nárast výkonnosti paralelného programu. Naopak, verejnú sekciu triedy sme rozšírili o deklaračné príkazy bezparametrického konštruktora a parametrickej výpočtovej metódy.

2. Obsah zdrojového súboru Class1.cpp zmeníme podľa tohto vzoru:

```
// Class1.cpp
#include "pch.h"
#include "Class1.h"

using namespace winrt_cx;
using namespace Platform;

// Definícia bezparametrického konštruktora.
ParalelneVypocty::ParalelneVypocty()
```

```
{
    narastVykonnosti = 0.0f;
}
// Definícia výpočtovej metódy.
float ParalelneVypocty::VypocitatNV(float tS, float tP, int n)
{
    narastVykonnosti = (tS + tP) / (tS + tP / n);
    return narastVykonnosti;
}
```

Zdrojový kód implementačného súboru je dobre pochopiteľný: prototypy funkčných členov uvedených v deklarácii primárnej odkazovej triedy sme rozvinuli, čím sme získali ich kompletné definície. Ako vyplýva z Amdahlovho zákona, koeficient určujúci nárast výkonnosti paralelného programu v pomere k výkonnosti sekvenčného programu vypočítame tak, že delíme celkové časy na spracovanie sekvenčných a paralelných algoritmov týchto programov na jedno- a viacprocesorových počítačoch.

3. Uskutočníme preklad projektu WinRT-komponentu (**Build → Build Solution**).

3.5.2 Použitie WinRT-komponentu jazyka C++/CX v prostredí WinRT-programu jazyka C#

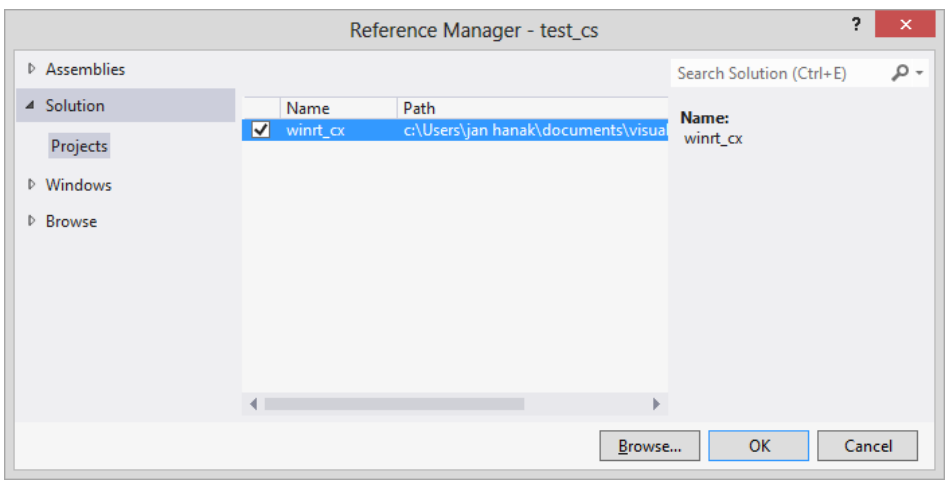
Hoci by sme mohli vytvoriť samostatne pôsobiaci nový projekt WinRT-programu v jazyku C#, lepším riešením je pridať nový projekt WinRT-programu do existujúceho riešenia jazyka C++/CX. To spravíme takto:

1. V podokne **Solution Explorer** klepneme pravým tlačidlom myši na názov riešenia.
2. V miestnej ponuke ukážeme na položku **Add** a uskutočníme selekciu príkazu **New Project**.
3. V stromovej štruktúre dialógu **Add New Project** zvolíme cestu **Other Languages → Visual C# → Windows Store**.

4. Zo súpravy projektových šablón vyberieme šablónu **Blank App (XAML)** a novému projektu prisúdime identifikátor test_cs.
5. Napokon klikneme na tlačidlo **OK**.

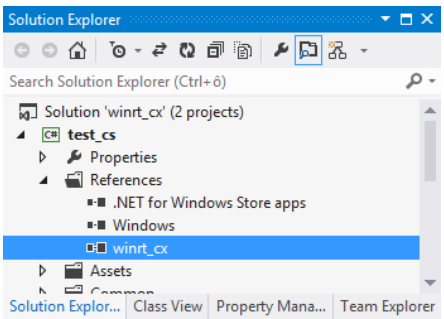
Ihneď, ako sprievodca pridá projekt WinRT-programu do existujúceho riešenia, prikážeme, aby úlohu štartovacieho projektu plnil práve tento nový projekt. Túto modifikáciu realizujeme klepnutím na názov projektu WinRT-programu a aktiváciou voľby **Set as StartUp Project** z miestnej ponuky. (Dôkazom správnosti nášho postupu je zvýraznenie názvu WinRT-programu tučným typom písma.)

Naším cieľom je využitie schopností WinRT-komponentu jazyka C++/CX z prostredia WinRT-programu jazyka C#. Aby sme mohli využívať primárnu odkazovú triedu **ParalelneVypocty** a jej metódu, musíme do projektu WinRT-programu jazyka C# vložiť odkaz na WinRT-komponent jazyka C++/CX. Otvoríme preto ponuku **Project** a klepneme na položku **Add Reference**. V dialógovom okne **Reference Manager** rozvinieme položku **Solution** a označíme položku **Projects**. Proces dokončíme aktiváciou WinRT-komponentu a klepnutím na tlačidlo **OK** (obr. 3.17).



Obr. 3.17: Import odkazu na WinRT-komponent jazyka C++/CX

Vykonanú importnú operáciu reflektuje aj zmena vzhľadu podokna **Solution Explorer**: do priečinka **References** bol pridaný uzol s názvom projektu WinRT-komponentu **winrt_cx** (obr. 3.18).



Obr. 3.18: Verifikácia korektnosti importu odkazu na cieľový WinRT-komponent

Na hlavnú stránku WinRT-programu jazyka C# pridáme jedno tlačidlo (objekt ovládacieho prvku **Button**) a jedno textové návestie (objekt ovládacieho prvku **TextBlock**). Logika, ktorú zamýšľame implementovať, bude prostá: po klepnutí na tlačidlo sa v textovom návesti objaví vypočítaný koeficient nárastu výkonnosti paralelného programu.

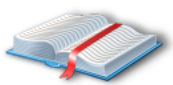
Syntaktický obraz spracovateľa udalosti **Click** tlačidla upravíme podľa nasledujúceho predpisu:

```
private void btnNV_Click(object sender, RoutedEventArgs e)
{
    // Inštanciácia primárnej odkazovej triedy WinRT-komponentu.
    winrt_komponent.ParalelneVypocty obj =
        new winrt_komponent.ParalelneVypocty();
    // Využitie funkcionality WinRT-komponentu.
    txbNV.Text = obj.VypocitatNV(40, 60, 8).ToString();
}
```

Komentár k zdrojovému kódu jazyka C#: Využitie schopností WinRT-komponentu jazyka C++/CX je v prostredí WinRT-programu jazyka C# veľmi rýchle a syntakticky elegantné.

V skutočnosti musíme urobiť len dve veci:

1. Inštanciovať primárnu odkazovú triedu WinRT-komponentu.
2. Aktivovať parametrickú metódu zostrojenej inštancie primárnej odkazovej triedy WinRT-komponentu.



Poznámka: Všimnime si, že vo výpise zdrojového kódu jazyka C# používame kvalifikovaný názov primárnej odkazovej triedy. Skrátenie zápisu dosiahneme tak, že direktívou **using** zavedieme cieľový menný priestor **winrt_komponent**. Potom stačí, keď sa na primárnu odkazovú triedu budeme odkazovať len pomocou jej názvu (bez špecifikácie príslušného menného priestoru).

Len čo sme vykonali všetky kroky naznačeného postupu, uskutočníme zostavenie riešenia. V ďalšej etape spustíme WinRT-program jazyka C# a otestujeme, či funguje tak, ako má.

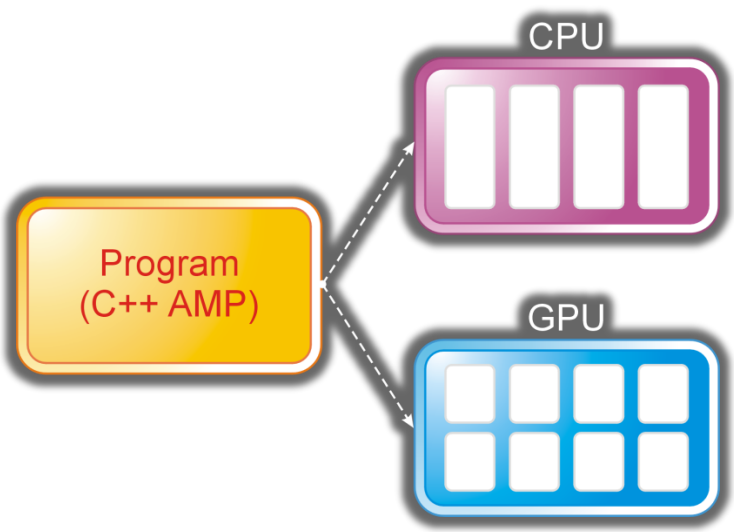
3.6 Heterogénny paralelizmus s platformou C++ AMP

S nástupom viacjadrových mikroprocesorov do komerčnej sféry sa začala paradigma paralelného programovania vo väčšej miere dotýkať aj bežných praktických vývojárov a programátorov. Cieľom paralelného programovania je využiť maximálnu výpočtovú kapacitu hocako dimenzovaného počítačového systému. Ak pri implementácii paralelných výpočtových modelov využívame len jeden druh výpočtových akcelerátorov počítača, hovoríme o tzv. homogénnom paralelizme. Homogénny paralelizmus sa napríklad uplatňuje pri vývoji paralelných programov, ktoré dokážu alokovať všetky osadené mikroprocesory, respektíve všetky výpočtové jadrá osadných mikroprocesorov. Za týchto okolností sú akcelerátormi výpočtových procesov viaceré výpočtové jadrá prítomných mikroprocesorov.

Moderné počítačové systémy sú však hardvérovo natoľko vyspelé, že okrem kolekcie výpočtových jadier mikroprocesora zahŕňajú aj súpravu výpočtových jadier grafického čipu (osadeného na grafickom akcelerátore). Tieto grafické jadrá sa vyskytujú v oveľa

vyššej početnosti ako jadrá mikroprocesora. Hoci mikroprocesory bežných počítačov obsahujú dve až štyri výpočtové jadrá, na grafickom čipe nájdeme stovky ba až tisíce výpočtových jadier. Tie sú vhodné najmä na riešenie dátovo-paralelných úloh. Ak pri implementácii paralelných výpočtových modelov upotrebujeme viacero druhov výpočtových akcelerátorov počítača, vravíme o tzv. heterogénnom paralelizme. V priamej konfrontácii s homogénnym paralelizmom teda môžeme vyhlásiť, že heterogénny paralelizmus dokáže využiť jednak výpočtové jadrá všetkých osadených mikroprocesorov a rovnako aj výpočtové jadrá všetkých osadených grafických akcelerátorov. Vďaka tomuto prístupu sme schopní diverzifikovať realizáciu výpočtových procesov paralelného programu na súpravu mikroprocesorov a grafických čipov.

Spoločnosť Microsoft prichádza s novou paralelnou platformou, ktorá sa volá C++ AMP (angl. *Accelerated Massive Parallelism*). Paralelná platforma C++ AMP integruje nový výpočtový model akcelerovaného masívneho paralelizmu, ktorý je optimalizovaný pre efektívnu implementáciu heterogénnych paralelných algoritmov. Základná funkcionalita paralelného programu, ktorý zavádza heterogénny paralelizmus, je dokumentovaná na obr. 3.19.



Obr. 3.19: Základná funkcionalita paralelného programu s C++ AMP

Záver

Vážení vývojári, programátori a softvéroví experti,

ďakujeme vám, že ste s nami absolvovali teoreticko-praktický kurz inovácií v programovacích jazykoch Visual Basic 2012, C# 5.0 a C++. Veríme, že sa tejto knihe podarilo splniť jej vzdelávací cieľ, ktorý bol sústredený na hĺbkový rozbor vybratých syntakticko-sémantických noviniek, zakomponovaných do spomenutej trojice programovacích jazykov.

Keďže softvérový svet je vskutku turbulentný, uvádzame na záver ďalšie hodnotné elektronické zdroje, z ktorých môžu vývojári pracujúci v jazykoch Visual Basic 2012, C# 5.0 a C++ čerpať nové poznatky:

1. Vývojárske stredisko pre programovací jazyk Visual Basic 2012.
   Adresa: http://msdn.microsoft.com/en-us/vstudio/hh388573.

2. Vývojárske stredisko pre programovací jazyk C# 5.0.
   Adresa: http://msdn.microsoft.com/en-us/vstudio/hh341490.

3. Vývojárske stredisko pre programovací jazyk C++.
   Adresa: http://msdn.microsoft.com/en-us/vstudio/hh386302.

4. Vývojárske stredisko pre natívne paralelné programovanie v jazyku C++.
   Adresa: http://msdn.microsoft.com/library/vstudio/hh875062(VS.110).aspx.

5. Vývojárske stredisko pre riadené paralelné programovanie v jazykoch Visual Basic 2012 a C# 5.0.
   Adresa: http://msdn.microsoft.com/library/vstudio/dd460693(VS.110).aspx.

Prajeme vám veľa úspechov pri vývoji moderných aplikácií v prostredí Visual Studio 2012!
Autor a realizačný tím spoločnosti Microsoft Slovakia

O autorovi

Ing. Ján Hanák, PhD., MVP, vyštudoval Ekonomickú univerzitu v Bratislave. Tu, na Katedre aplikovanej informatiky Fakulty hospodárskej informatiky (KAI FHI), pracuje ako vysokoškolský pedagóg. Prednáša a vedie semináre týkajúce sa programovania a vývoja počítačového softvéru v programovacích jazykoch C, C++ a C#. K jeho favoritom patrí tiež Visual Basic, F# a C++/CLI.

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

1. **Inovácie v prostredí Visual Studio 2012**. Bratislava: Microsoft Slovakia, 2012.
2. **Vývoj moderných WinRT-programov pre systém Windows 8**. Bratislava: Microsoft Slovakia, 2012.
3. **Základy databázového vývoja v prostredí Visual Studio LightSwitch 2011**. Bratislava: Microsoft Slovakia, 2012.
4. **Visual Basic 2010 - praktické příklady**. Kralice na Hané: Computer Media, 2012.
5. **Moderné paralelné programovanie**. Bratislava: Vydavateľstvo EKONÓM, 2011.
6. **Programování v jazyce Visual Basic 2010**. Kralice na Hané: Computer Media, 2011.
7. **Softvérové technológie na platforme Microsoft .NET**. Bratislava: Eurokódex, 2011.
8. **Rýchly vývoj aplikácií v jazyku Visual Basic 2010 pre systém Windows 7**. Bratislava: Microsoft Slovakia, 2011.
9. **Programování v jazyce C**. Kralice na Hané: Computer Media, 2011.
10. **Ako sa stať softvérovým vývojárom**. Bratislava: Microsoft Slovakia, 2010.
11. **C++: Akademický výučbový kurz**. Bratislava: Vydavateľstvo EKONÓM, 2010.
12. **Inovácie v jazykoch Visual Basic 2010, C# 4.0 a C++**. Brno: Artax, 2010.
13. **Programování v jazyce C**. Kralice na Hané: Computer Media, 2010.
14. **Visual Basic 2010 – Hotové riešenia**. Bratislava: Microsoft Slovakia, 2010.

15. **C#: Akademický výučbový kurz, 2. aktualizované a rozšírené vydanie**. Bratislava: Vydavateľstvo EKONÓM, 2010.
16. **Praktické objektové programování v jazyce C# 4.0**. Brno: Artax, 2009.
17. **Praktické paralelné programovanie v jazykoch C# 4.0 a C++**. Brno: Artax, 2009.
18. **C++/CLI - Praktické príklady**. Brno: Artax, 2009.
19. **C# 3.0 - Programování na platformě .NET 3.5**. Brno: Zoner Press, 2009.
20. **C++/CLI - Začínáme programovat**. Brno: Artax, 2009.
21. **C#: Akademický výučbový kurz**. Bratislava: Vydavateľstvo EKONÓM, 2009.
22. **Základy paralelného programovania v jazyku C# 3.0**. Brno: Artax, 2009.
23. **Objektovo orientované programovanie v jazyku C# 3.0**. Brno: Artax, 2008.
24. **Inovácie v jazyku Visual Basic 2008**. Praha: Microsoft, 2008.
25. **Visual Basic 2008: Grafické transformácie a ich optimalizácie**. Bratislava: Microsoft Slovakia, 2008.
26. **Programovanie B – Zbierka prednášok (Učebná pomôcka na programovanie v jazyku C++)**. Bratislava: Vydavateľstvo EKONÓM, 2008.
27. **Programovanie A – Zbierka prednášok (Učebná pomôcka na programovanie v jazyku C)**. Bratislava: Vydavateľstvo EKONÓM, 2008.
28. **Expanzívne šablóny: Príručka pre tvorbu "code snippets" pre Visual Studio**. Bratislava: Microsoft Slovakia, 2008.
29. **Kryptografia: Príručka pre praktické odskúšanie symetrického šifrovania v .NET Framework-u**. Bratislava: Microsoft Slovakia, 2007.
30. **Príručka pre praktické odskúšanie vývoja nad Windows Mobile 6.0**. Bratislava: Microsoft Slovakia, 2007.
31. **Príručka pre praktické odskúšanie vývoja nad DirectX**. Bratislava: Microsoft Slovakia, 2007.
32. **Príručka pre praktické odskúšanie automatizácie aplikácií Microsoft Office 2007**. Bratislava: Microsoft Slovakia, 2007.
33. **Visual Basic 2005 pro pokročilé**. Brno: Zoner Press, 2006.
34. **C# – praktické příklady**. Praha: Grada Publishing, 2006.
35. **Programujeme v jazycích C++ s Managed Extensions a C++/CLI**. Praha: Microsoft, 2006.

36. **Přecházíme z jazyka Visual Basic 6.0 na jazyk Visual Basic 2005**. Praha: Microsoft, 2005.
37. **Visual Basic .NET 2003 – Začínáme programovat**. Praha: Grada Publishing, 2004.

V rokoch 2006 – 2012 bol jeho prínos vývojárskym komunitám ocenený celosvetovými vývojárskymi titulmi **Microsoft Most Valuable Professional (MVP)** s kompetenciou **Visual Developer – Visual C++**.

Použitá literatúra

1. CAMPBELL, C. – JOHNSON, R. – MILLER, A. – TOUB, S.*: Parallel Programming with Microsoft .NET (Design Patterns for Decomposition and Coordination on Multicore Architectures)*. Redmond: Microsoft Press, 2010. ISBN 978-0-7356-5159-3.

2. GREGORY, K. – MILLER, A.: *C++ AMP (Accelerated Massive Parallelism with Microsoft Visual C++)*. Sebastopol, California: O'Reilly Media, Inc., 2012. ISBN 978-0-7356-6473-9.

3. MARSHALL, D.: *Parallel Programming with Microsoft Visual Studio 2010*. Sebastopol, California: O'Reilly Media, Inc., 2011. ISBN 978-0-7356-4060-3.

4. MIDKIFF, S. P.: *Automatic Parallelization (An Overview of Fundamental Compiler Techniques)*. San Rafael, California: Morgan&Claypool Publishers, 2012. ISBN 978-1-60845-841-7.

Ing. Ján Hanák, PhD., MVP
pôsobí ako vysokoškolský pedagóg, spisovateľ počítačovej literatúry, softvérový vývojár a evanjelista moderných IT technológií. Na svojom konte má takmer 40 odborných prác, ku ktorým patria vysokoškolské učebnice, vedecké a odborné monografie, skriptá a príručky venované problematike vývoja počítačového softvéru v programovacích jazykoch C, C++, C#, Visual Basic a C++/CLI. V rokoch 2006 – 2012 bol jeho prínos vývojárskym komunitám ocenený celosvetovo uznávanými vývojárskymi titulmi spoločnosti Microsoft s názvom Most Valuable Professional (MVP) s kompetenciou Visual Developer – Visual C++.

ISBN 978-80-87017-11-1