Microsoft
Vaše možnosti. Naše inspirace.™

Ján Hanák

Programujeme v jazycích C++ s Managed Extensions a C++/CLI

Microsoft Visual Studio .net

Microsoft Visual Studio 2005

Programujeme v jazycích C++ s Managed Extensions a C++/CLI

Ján Hanák, 2006

Obsah

Úvod

Po razantním nástupu vývojově-exekuční platformy .NET Framework společnosti Microsoft se mnozí vývojáři a programátoři domnívali, že tandem nejlepších programovacích nástrojů pro vývoj skutečných aplikací .NET tvoří jazyky Visual Basic .NET a Visual C# .NET. Uvedený fenomén byl opravdu znatelný: Většina ukázkových aplikací, programátorských postupů a techničtěji zaměřených materiálů demonstrovala vykládanou problematiku na kódových konstrukcích jazyků Visual Basic .NET nebo C#. Jakoby ve své vlastní dimenzi ovšem zůstával vývojářsky komplet s názvem Visual C++ .NET, což byla skutečnost, která mohla v mnoha potenciálních zájemcích vyvolávat dojem, že C++ již není tím pravým vývojářským nástrojem pro budování řízených softwarových aplikací.

Hlavním existenčním smyslem této vývojářské příručky je zvrátit právě popsaný mýtus a ukázat vám, příznivcům jazyka C++, že nové reinkarnace tohoto jazyka, jež jsou ztělesněny v produktech C++ s Managed Extensions a C++/CLI, disponují úplnou a konkurenceschopnou výbavou pro vytváření náročných a sofistikovaných aplikací běžících pod křídly platformy Microsoft .NET Framework verze 1.1 a 2.0.

Vývojáři, kteří investovali do studia tajů jazyka C++ nemálo času a energie, jistě rádi uslyší, že jejich úsilí nebylo v žádném případě zbytečné. Nabyté znalosti a zkušenosti jim budou jenom k užitku, i když je nutno současně dodat, že problematika vývoje řízených aplikací .NET jako taková si vyžádá další studium. Jestliže se ptáte proč, máme pro vás jednoduchou odpověď. Tak předně, řízené C++ je ve skutečnosti natolik velkolepým rozšířením tradičního (nativního) C++, že můžeme bez jakýchkoliv obav mluvit o zcela novém programovacím jazyku. Situace se však poněkud komplikuje, když prohlásíme, že pod termínem „řízené C++" můžeme v současné době chápat dvě vývojářská prostředí. To první je známé jako C++ s Managed Extensions a bylo uvedeno již s první verzí „dotnetového" Visual Studia, tedy Visual Studia .NET 2002. Na tuto verzi řízeného C++ se budeme v této publikaci odkazovat jako na C++ s Managed Extensions. Řízené C++ v této podobě se zanedlouho v prakticky nezměněném stavu objevilo i ve Visual Studiu .NET 2003 v podobě produktu Visual C++ .NET 2003. Jazyk C++ s Managed Extensions ve vyhotovení pro rok 2003 je proto takřka identický s verzí 2002. Verze 2003 je přesto něčím výjimečná: tím pomyslným diamantem byl vizuální návrhář v integrovaném vývojovém prostředí Visual C++ .NET 2003, jenž do těch nejmenších detailů naplňoval koncepci vizuálního programování. Vývojáři tak nebyli nuceni manuálně psát kód pro výstavbu grafických uživatelských rozhraní (GUI) svých aplikací, ale místo toho mohli všechny prvky jednoduše nakreslit, zcela stejně jako jejich kolegové pracující v jazycích Visual Basic nebo C#.

Druhé řízené C++ spatřilo světlo světa koncem roku 2005, kdy společnost Microsoft vypustila na trh produkt Visual C++ 2005 společně s novou vývojově-exekuční platformou .NET Framework 2.0. Pokud vlastníte zmíněný vývojářský nástroj, pak byste měli vědět, že řízené C++, které je v něm implementováno, se nazývá C++/CLI. Zkratka „CLI" v názvu odpovídá slovnímu spojení „Common Language Infrastructure", čímž dává najevo, že experti v Microsoftu ještě více vylepšili C++ s Managed Extensions a zahájili tak novou epochu jazyka C++ pro přípravu aplikací .NET nové generace. Jazyk C++/CLI je nástupcem dřívějšího C++ s Managed Extensions, no jedním dechem je zapotřebí říci, že se v žádném případě nejedná o pouhá kosmetická vylepšení a zdokonalení. Kdepak, vážení přátelé, C++/CLI přichází se zbrusu novou syntaxí a mnoha hlubokými inovacemi, díky nimž se stává flexibilnějším a produktivnějším nástrojem než kdykoliv předtím.

Když jsme projektovali obsahovou strukturu této příručky, chtěli jsme, aby dílo dovedli využít dvě skupiny vývojářů: jednak ti, kteří vlastní Visual Studio .NET 2003 a rádi by nakoukli do komnaty s nápisem C++ s Managed Extensions, a také ti, kteří již v jazyce C++ s Managed Extensions pracují, no chtěli by přejít na C++/CLI, respektive Visual Studio 2005. Přiznáváme, že jsme si stanovili nelehký úkol, no konec konců, co bychom

pro vás neudělali. Nuže, zatímco v první části publikace se setkáte s úvodem do programování v jazyce C++ s Managed Extensions, v druhém tematickém celku se pak společně podíváme na novinky, jež nás čekají v C++/CLI. Pevně věříme, že nastíněné rozvržení vám bude vyhovovat.

Typografické konvence

Abychom vám čtení této publikace zpříjemnili v co možná největší míře, byl přijat kodex typografických konvencí, jehož pomocí došlo ke standardizaci a unifikaci použitých textových a grafických symbolů a stylů. Věříme, že přijaté konvence napomohou zvýšení přehlednosti a uživatelské přívětivosti výkladu. Přehled použitých typografických konvencí je uveden v tab. 1.

Tab. 1: Přehled použitých typografických konvencí

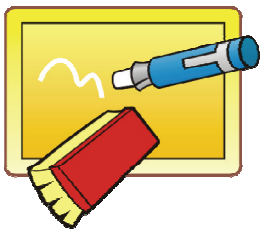
Typografická konvence	Ukázka použití typografické konvence
Veškerý text, který neoznačuje zdrojový kód, jména identifikátorů a klíčových slov jazyků C++ s Managed Extensions a C++/CLI, ani názvy jiných programových elementů a entit, je psán standardním písmem.	Vývojově-exekuční platforma Microsoft .NET Framework 2.0 vytváří společně s jazykem C++/CLI jednotné zázemí pro vytváření moderních aplikací .NET pro operační systémy řady Windows.
Názvy nabídek, položek nabídek, ovládacích prvků, komponent, dialogových oken, podpůrných softwarových nástrojů, typů projektů, jakožto i názvy dalších součástí grafického uživatelského rozhraní jsou formátovány **tučným písmem**.	Pro založení nového projektu standardní aplikace pro Windows (**Windows Forms Application**) ve Visual C++ 2005 postupujte následovně: 1. Otevřete nabídku **File**, ukažte na položku **New** a klepněte na příkaz **Project...**. 2. V dialogovém okně **New Project** klepněte ve stromové struktuře **Project Types** na položku **Visual C++**. 3. Ze sady projektových šablon (**Templates**) vyberte ikonu **Windows Forms Application**. 4. Do textového pole **Name** zapište název pro novou aplikaci a stiskněte tlačítko **OK**.
Klávesové zkratky a jejich kombinace jsou uváděny KAPITÁLKAMI.	Jestliže chcete otevřít již existující projekt jazyka C++/CLI, použijte klávesovou zkratku CTRL+SHIFT+O.

Fragmenty zdrojových kódů jazyků C++ s Managed Extensions a C++/CLI, případně také jiných programovacích jazyků, jsou formátovány neproporcionálním písmem `Courier New`. Kromě toho jsou ve všech výpisech barevně rozlišeny následující programové elementy:
1. Klíčová slova programovacího jazyka jsou formátována modrou barvou.
2. Komentáře, které blíže popisují charakter činnosti programového příkazu nebo bloku programových příkazů, jsou zobrazeny pomocí zelené barvy.
3. Veškerý ostatní kód, jenž neoznačuje ani klíčová slova, ani komentáře, je formátován standardní černou barvou.
	`// Vytvoření instance gc třídy.`
	`GC_Tridy::A __gc * p_A = __gc new GC_Tridy::A();`
	`// Použití objektu pro získání`
	`// hodnoty manipulátoru.`
	`MessageBox::Show(String::Concat (S"Manipulátor pracovní plochy: ", p_A->Ziskat_Manipulator()));`
	`// Explicitní dealokace nativních`
	`// i řízených zdrojů objektu.`
	`p_A->Dispose();`

Neproporcionální písmo `Courier New` je kromě výpisů zdrojových kódů použito také při uvádění názvů programových entit v základním výkladovém textu. Tímto stylem písma jsou formátovány například názvy tříd a jejich instancí, názvy proměnných a metod a rovněž tak i názvy jiných programových identifikátorů.	Statická metoda `Sleep` třídy `Thread` z jmenného prostoru `System::Threading` dovede uspat aktivní programové vlákno na specifikovaný počet milisekund (ms). Když metodě `Sleep` předáte celočíselnou hodnotu udávající počet milisekund pro uspání programového vlákna, metoda zabezpečí bezpečné uspání vlákna po určenou dobu.

Vyjma typografických konvencí uvedených v tab. 1 se můžete v textu vývojářské příručky setkat také s informačními ikonami, které vám poskytují hodnotné informace související s právě probíranou problematikou. Výčet informačních ikon můžete vidět v tab. 2.

Tab. 2: Přehled informačních ikon

Informační ikona	Název informační ikony	Charakteristika
	**Upozornění**	Upozorňuje čtenáře na důležité skutečnosti, které by měl mít v každém případě na paměti, neboť na nich může záviset pochopení dalších souvislostí nebo úspěšné provedení postupu či pracovního algoritmu.
	**Poznámka**	Sděluje čtenáři další a podrobnější informace, které se pojí s vykládanou tematikou. Ačkoliv je míra důležitosti této informační ikony nižší než výše uvedené ikony, ve všeobecnosti se doporučuje, aby čtenář věnoval doplňujícím informačním sdělením svoji pozornost. Může se tak dozvědět nová fakta, nebo najít skryté souvislosti mezi již známými poznatky.
	**Tip**	Poukazuje na lepší, efektivnější nebo rychlejší splnění programovacího úkolu či postupu. Uvidí-li čtenář v textu publikace tuto informační ikonu, může si být jist, že nalezne jedinečný a prověřený způsob, jak produktivněji dosáhnout kýženého cíle.

Informační ikony vystupují jako samostatné ostrůvky, které vám nabízejí relevantní informace z oblasti programování v jazycích C++ s Managed Extensions a C++/CLI. Při jejich tvorbě byly brány v potaz následující aspekty:

1. Informační ikony musejí byt schopny upoutat pozornost čtenáře, a to zejména v okamžiku, kdy je nutné podat vysvětlení obzvlášť důležitého pojmu, termínu nebo technologie. Kromě toho je úkolem informačních ikon přinášet dodatečné poznatky a poukazovat na možnosti efektivnějšího vyřešení programátorského problému či postupu.

2. Informační ikony musejí dodržovat standardní linii výkladu. Jedině tak je zabezpečeno, že čtenář bude moci vykládané skutečnosti okamžitě využít ve svůj prospěch.

3. Informační ikony musejí být hezky graficky vyvedeny, aby byly oku lahodící a dokázaly tak přispět k zvýšení uživatelského komfortu publikace.

Poděkování

Na tomto místě bych velice rád vyjádřil své srdečné díky všem lidem, kteří se zasloužili o to, abyste mohli v rukou držet tuto vývojářskou příručku. Mé veliké poděkování a uznání patří především dvěma skvělým pracovníkům společnosti Microsoft, jmenovitě pánům Jiřímu Burianovi a Miroslavu Kubovčíkovi, jejichž ochota, vstřícnost a nadšení pro věc se snad ani nedá popsat slovy. Řečeno stručně a jasně, byla radost s nimi spolupracovat. Mé díky si však zaslouží i další kolegové, ať už grafici nebo sazeči, kteří přidali ruku k dílu a přispěli tak k naší společné snaze předložit vám poutavou publikaci o programování v jazycích C++ s Managed Extensions a C++/CLI. Takže ještě jednou: vřelé díky!

Část 1 - Programovací jazyk C++ s Managed Extensions

Začínáme s jazykem C++ s Managed Extensions

Programovací jazyk C++ s Managed Extensions se vyskytuje ve dvou vydáních softwarového produktu Visual C++ .NET (2002 a 2003). My budeme pracovat s implementací s pořadovým číslem 2003. Po pravdě řečeno, Visual C++ .NET 2003 je vskutku prvotřídní komplet, jehož pomocí mohou vývojáři, programátoři a softwaroví architekti budovat široké spektrum počítačových aplikací. Jestliže jste si pořídili Visual C++ .NET 2003, smíte aplikovat následující vývojářské přístupy:

1. Vytváření nativních (neřízených) aplikací založených na Win32 API a jazyku C.

2. Vytváření nativních (neřízených) aplikací v jazyce C++ za asistence knihovny tříd MFC (Microsoft Foundation Classes).

3. Vytváření řízených aplikací pro vývojově-exekuční platformu Microsoft .NET Framework 1.1 prostřednictvím řízeného C++, tedy jazyka C++ s Managed Extensions.

Je zřejmé, že náš zájem se bude soustředit zejména na poslední variantu. Ještě před tím, než se budeme moci zaměřit na vysvětlení základů řízeného C++, musíme si alespoň ve stručnosti představit pracovní prostředí .NET Framework 1.1 jakožto i stěžejní komponenty, na nichž je tato platforma založena.

Charakteristika řízeného prostředí platformy Microsoft .NET Framework 1.1

Microsoft .NET Framework 1.1 reprezentuje vývojové a exekuční prostředí, které bylo navrženo speciálně pro tvorbu a běh .NET-kompatibilních počítačových aplikací. Takovéto zaměření platformy je svým způsobem jedinečné, neboť nízkoúrovňové softwarové služby, jež pohánějí celou tuto ohromnou mašinérii, musejí být přítomny nejenom na každé vývojářské počítačové stanici, ale také na každém klientském PC, na němž má být požadovaná řízená aplikace spuštěna. Platforma .NET Framework 1.1 je složena z několika stěžejních pilířů, na které se nyní blíže podíváme.

1. **Společné běhové prostředí (Common Language Runtime, CLR).** Společné běhové prostředí nebo také běhové prostředí CLR má na starosti korektní exekuci programového kódu aplikací .NET. Veškerý zdrojový kód řízených aplikací je po jejich sestavení přeložen do podoby kódu mezijazyka, jenž nese jméno Microsoft Intermediate Language (MSIL nebo též jenom IL). Stručně řečeno, MSIL je objektově orientovaný jazyk, který podporuje jazykovou interoperabilitu na nízké úrovni. Kód tohoto jazyka ovšem není přímo zpracováván instrukční sadou mikroprocesoru, ale na požádání dochází k jeho překladu do nativního kódu díky Just-In-Time (JIT) kompilátoru. Výsledným produktem JIT překladače je tedy nativní kód, jenž vznikl v procesu překladu kódu jazyka MSIL. Takto získaný nativní kód již může být nabídnut CPU a podroben tak přímé exekuci. Společné běhové prostředí nabízí řízeným aplikacím své služby také v dalších oblastech, k nimž patří především realizace automatické správy paměti, dohled na efektivnější politikou správy verzí, produktivnější distribuce aplikačních jednotek anebo rozproudění jazykové interoperability za pomocí technologií P/Invoke, COM Interop či IJW (It Just Works).

2. **Bázová knihovna tříd (.NET Framework Class Library, FCL).** Bázová knihovna tříd je téměř nevyčerpatelnou studnicí řízených datových typů, tříd, struktur, enumerací a delegátů, které mohou vývojáři při vytváření svých aplikací využít (mimochodem, počet dostupných programových elementů se rátá na

tisíce). Řízené třídy v přívětivé formě zapouzdřují funkcionalitu mnoha nativních funkcí aplikačního programového rozhraní operačního systému Windows. Řečeno kvantitativně, bázová knihovna tříd je schopna uskutečnit přibližně 95 % všech aktivit, které bylo doposud nutné provádět buď přes „čisté" Win32 API nebo MFC. Vestavěné třídy jsou hierarchicky uspořádány do logických jmenných prostorů, což na straně jedné pozitivně podporuje přehlednost a orientační schopnost, zatímco na straně druhé minimalizuje vznik jmenných konfliktů jednotlivých programových identifikátorů. Instance řízených tříd jsou pod neustálým dohledem automatické správy paměti běhového prostředí CLR platformy Microsoft .NET Framework 1.1. Třídy lze opětovně používat, a tudíž lze od jedné specifické třídy odvodit třídu novou a vzápětí překrýt nebo zastínit vybrané metody či vlastnosti bázové třídy (samozřejmě za předpokladu, že mateřská třída není definována jako zapečetěná, protože v tomto případě by generování podtříd nebylo realizovatelné).

3. **Společný typový systém (Common Type System, CTS).** Společný typový systém tvoří soustava pravidel, jež explicitně deklarují požadavky a standardy kladené na .NET-kompatibilní datové typy. Všechny zabudované i uživatelsky definované datové typy musejí vyhovovat kritériím typové bezpečnosti, výkonnosti a jazykové interoperability. Mimořádná péče se soustředí převážně na jazykovou interoperabilitu, neboť jedním z cílů platformy .NET Framework 1.1 je rovněž bezproblémový vývoj vzájemně kooperujících řízených aplikačních modulů a jejich částí. Ačkoliv datové typy můžeme dělit podle více hledisek, nejčastěji dochází k jejich kategorizaci dle charakterových vlastností na hodnotové a odkazové (neboli referenční) datové typy. Typy však pochopitelně můžeme třídit i podle jejich vztahu k bázové knihovně tříd na primitivní (kupříkladu `char`, `int` a `long`) a uživatelsky definované (třídy, struktury, delegáty, enumerace atd.).

4. **Společná jazykové specifikace (Common Language Specification, CLS).** Společná jazyková specifikace determinuje požadavky na .NET-kompatibilní programovací jazyky a jejich kompilátory, které si poradí s generováním zdrojových instrukcí mezijazyka MSIL. Specifikace CLS ve velké míře spolupracuje s pravidly, které definuje společný typový systém (CTS). Synergickým efektem se pak stává garance typové bezpečnosti a jazykové interoperability. Řečeno jinými slovy, jakýkoliv programovací jazyk splňující požadavky společné jazykové specifikace může být označen přívlastkem „vhodný pro platformu .NET". Tím pádem je zaručeno, že aplikace připravená v tomto jazyce bude moci vést vzájemný komunikační dialog s kteroukoliv jinou aplikací také napsanou v .NET-kompatibilním programovacím jazyce. Požadavky společné jazykové specifikace tak představují společnou průnikovou množinu programových konstrukcí, které musí implementovat každý jazyk, jenž se rozhodne dotyčné specifikaci vyhovět.

Dosud byly uvedeny tři verze vývojově-exekuční platformy Microsoft .NET Framework. V únoru 2002 byla představena počáteční verze 1.0, kterou za více než rok následovala verze s inkrementálním označením 1.1 (stalo se tak v dubnu 2003). Nejnovější přírůstek nese jméno .NET Framework 2.0 a mezi vývojáře pronikl na sklonku roku 2005. Jednou z předností platformy .NET Framework je její schopnost paralelní koexistence její více verzí. Tím chceme říci, že na jedné počítačové stanici se mohou v jednom okamžiku nacházet všechny tři verze této platformy (1.0, 1.1 a 2.0). Aplikace .NET přitom vědí, kterou verzi společného běhového prostředí použít. Pravidlem je, že se použije ta verze, pro níž byla aplikace navržena a vyvinuta. Řízené aplikace jsou však ve spojení s běhovým prostředím CLR natolik inteligentní, že jim nečiní potíže ani navázání náležité komunikace s více verzemi externích programových součástí, komponent a dynamicky linkovaných knihoven. Bohudík, pryč jsou tak doby, kdy programátory strašila představa „pekla knihoven DLL".

Produkt Visual C++ .NET 2003 a aplikace .NET

Visual C++ .NET 2003 je kvalitním vývojářským nástrojem, jehož pomocí mohou zkušení programátoři v C++ připravovat nativní i řízené aplikace. Je zcela pochopitelné, že v našem zorném poli budeme preferovat právě řízené aplikace, tedy aplikace pro platformu .NET, jež jsou spravovány společným běhovým prostředím CLR. Jistě se však zajímáte o to, jaké typy aplikací můžete v jazyce C++ s Managed Extensions napsat. V tab. 1.1 jsme pro vás připravili charakteristiku vestavěných projektových šablon, s nimiž se setkáte v prostředí Visual C++ .NET 2003.

Tab. 1.1: Popis aplikací .NET, které lze připravit ve Visual C++ .NET 2003

Pojmenování aplikace .NET	Ikona aplikačního projektu	Charakteristika
ASP.NET Web Service (ASP.NET Webová služba)	 ASP.NET Web Service	XML webová aplikace, která využívá řízená rozšíření jazyka C++. Tento typ aplikace může být nasazen na webovém serveru, přičemž disponuje kompetencemi na vedení vzdálené komunikace s různými kategoriemi klientských aplikací.
Class Library (.NET) (Knihovna tříd)	 Class Library (.NET)	Knihovna tříd zapouzdřuje soustavu řízených tříd, které implementují požadovanou funkcionalitu. Instance těchto tříd mohou být vytvářeny i z prostředí jiných .NET-kompatibilních programovacích jazyků. Programový kód knihovny tříd je po překladu převeden do formy MSIL kódu, jenž je následně uložen do souboru s extenzí .dll. Knihovnu tříd si můžete představit jako řízenou alternativu klasických knihoven DLL, pouze s tím rozdílem, že řízená varianta je mnohem programátorsky přívětivější.
Console Application (.NET) (Konzolová aplikace)	 Console Application (.NET)	Konzolová aplikace je typická tím, že patří do skupiny těch aplikací, které pracují v prostředí příkazového řádku. Pro tento druh aplikací je společná absence grafického uživatelského rozhraní, nakolik aplikace samotná pracuje pouze v textovém módu, čímž je její rozhraní omezeno jenom na zobrazování alfanumerických textových znaků. Navzdory tomu je konzolová aplikace užitečným pomocníkem při mnoha příležitostech: tak třeba při zobrazování systémových zpráv či analýze záznamů a podobně.
Windows Control Library (.NET) (Knihovna ovládacích prvků)	 Windows Control Library (.NET)	Pokud se rozhodnete upotřebit možnosti jazyka C++ s Managed Extensions při vývoji ovládacích prvků platformy .NET pracujících na báze knihovny Windows Forms, zcela určitě si vyberete tuto projektovou šablonu. Z formálního hlediska můžeme ovládací prvek definovat jako objektově orientovanou kolekci programového kódu a dat, která vykonává jistou činnost. Vývoj uživatelských ovládacích prvků je citelně usnadněn začleněním vizuálního návrháře, kterého si bezesporu oblíbíte.
Windows Forms Application (.NET) (Standardní aplikace pro systém Windows)	 Windows Forms Application (.NET)	Přestože platforma Microsoft .NET Framework 1.1 dovoluje programátorům budovat široké portfolio řešení, mezi nimiž nechybějí XML webové služby či ASP.NET webové aplikace, hlavním vývojovým proudem pro nemálo vývojářů i nadále zůstávají standardní aplikace pro systém Windows s bohatým grafickým uživatelským rozhraním. Jsme rádi, že vám můžeme sdělit následující zprávu: Tvorba aplikací pro Windows je ve Visual C++ .NET 2003 zrovna tak intuitivní jak je tomu v jazycích Visual Basic .NET a C#.
Windows Service (.NET) (Služba systému Windows)	 Windows Service (.NET)	Služby systému Windows jsou poslední kategorií aplikací .NET, jež máte možnost pomocí řízeného C++ zhotovovat. Služby Windows reprezentují aplikace bez grafického uživatelského rozhraní, které pracují v pozadí. Jejich životní cyklus zpravidla kopíruje životní cyklus operačního systému, pod nímž tyto služby pracují. To znamená, že služby Windows se aktivují po nastartování systému a jejich životní pouť se končí těsně před ukončením činnosti systému.

I když je pravdou, že počet projektových šablon pro vytváření aplikací .NET se ve Visual C++ .NET 2003 zvýšil, přesto není v jazyce C++ s Managed Extensions možné připravovat tolik řízených aplikací jako třeba ve Visual Basicu .NET nebo v C#. V tab. 1.2 ukazujeme komparaci nástrojů Visual C# .NET 2003 a Visual C++ .NET 2003 z hlediska možnosti vývoje jednotlivých aplikací pro vývojově-exekuční platformu Microsoft .NET Framework 1.1.

Tab. 1.2: Porovnání dostupnosti vývoje rozličných typů aplikací .NET v produktech Visual C# . NET 2003 a Visual C++ .NET 2003

Aplikace jazyka Visual C# .NET 2003	Ekvivalent ve Visual C++ .NET 2003	
Windows Application	✓	Windows Forms Application (.NET)
Class Library	✓	Class Library (.NET)
Windows Control Library	✓	Windows Control Library (.NET)
Smart Device Application	✗	Žádný ekvivalent
ASP.NET Web Application	✗	Žádný ekvivalent
ASP.NET Web Service	✓	ASP.NET Web Service
ASP.NET Mobile Web Application	✗	Žádný ekvivalent
Web Control Library	✗	Žádný ekvivalent
Console Application	✓	Console Application (.NET)
Windows Service	✓	Windows Service (.NET)

Z přehledné tabulky můžeme snadno vyčíst, že ačkoliv si řízené C++ hravě poradí s většinou relevantních aplikací .NET, prozatím není zařazena přímá podpora pro vývoj řízených aplikací pro inteligentní mobilní zařízení (**Smart Device Application**), mobilní webové aplikace (**ASP.NET Mobile Web Application**), obecné webové aplikace (**ASP.NET Web Application**) a knihovny webových ovládacích prvků (**Web Control Library**).

Pro koho je jazyk C++ s Managed Extensions určen

Řízené C++ je novým programovacím jazykem, jehož dovednosti mohou využít níže uvedený skupiny vývojářů, programátorů a softwarových odborníků:

- **Programátoři znalí jazyka C++, kteří by rádi začali psát řízené aplikace pro platformu .NET Framework 1.1.** Jestliže se řadíte mezi vývojáře, kteří dobře ovládají nativní C++, přičemž byste chtěli přejít na moderní vývojovou platformu pro 21. století, jazyk C++ s Managed Extensions je pro vás tou správnou volbou. Řízená rozšíření, která byla jazyku C++ přisouzena do vínku, mají poměrně dalekosáhlé účinky, v důsledku čehož máte před sebou nový programovací stroj. Pakliže máte zkušenosti s nativním C++, bude pro vás přechod do řízeného prostředí rozhodně méně problematický, ačkoliv stále musíte počítat s tím, že skutečné ovládnutí řízeného C++ si bude vyžadovat jistou „programátorskou" pozornost.

- **Programátoři používající jazyk C s Win32 API a vývojáři píšící své aplikace pomocí C++ a knihovny tříd MFC.** O tom, že existuje početní skupina tvůrců softwaru, která se soustřeďuje na vývoj neřízených aplikací, a to buď pomocí „čistého" C a Win32 API, nebo prostřednictvím C++ s MFC, nemusíme dlouze diskutovat. Je to jednoduše fakt, který nesmíme přehlížet. Jelikož tito programátoři již napsali tisíce řádků zdrojového kódu, jejich pohled na možnou migraci do prostředí řízeného C++ bude patrně nejvíc skeptický. Nicméně i tak si dovolíme tvrdit, že také pro právě charakterizovaný trhový segment vývojářů je jazyk C++ s Managed Extensions vhodným řešením. Hoši z Redmondu totiž nezapomněli ani na tento fakt a do řízeného C++ začlenili pokročilé technologie pro vzájemnou spolupráci nativního a řízeného programového kódu. Na dosah ruky tak máte tři interoperabilní mechanizmy, které se pyšní názvy P/Invoke, COM Interop a It Just Works (IJW). Pokud je povoláte do boje, můžete i ze svých nových aplikací .NET volat fragmenty nativního kódu, které již existují hezkých pár let. Na druhou stranu, aplikace řízené běhovým prostředím CLR mají vstupenku, která jim umožňuje čerpat přehršel dalších výhod. Za všechny vzpomeňme například automatickou kontrolu životních cyklů vytvářených objektů a jejich zcela implicitní uvolňování z operační paměti v okamžiku, kdy se stanou nepotřebnými.

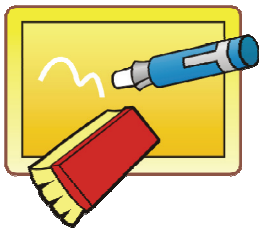
- **Vývojáři tvořící projektová řešení v jazycích Visual Basic .NET a C#, kteří mají zájem o ovládnutí pokročilých programátorských rysů a konstrukcí, jež nejsou dostupné v obou vzpomenutých jazycích.** Řízené C++ je otevřené také pro programátory ve Visual Basicu a C#. V této souvislosti je nutno podotknout, že vývojáři v jazyku C# budou mít k C++ s Managed Extensions přece jenom o něco blíž než jejich kolegové používající Visual Basic. Důvodem je skutečnost, že programovací jazyk C# je vyústěním evoluční řady jazyků z rodiny C/C++, zatímco Visual Basic .NET následuje svou vlastní linii. Jeden zásadní rozdíl spočívá kupříkladu v možnosti využití ukazatelů a referencí. V řízeném C++ mohou programátoři pracovat s těmito elementy bez jakýchkoliv omezení, což není v jazycích Visual Basic .NET a C# povoleno. (Abychom byli zcela přesní, tak dodejme, že C# dovoluje používat ukazatele pouze v blocích takzvaného nebezpečného kódu, které jsou vymezeny klíčovým slovem `unsafe`. Visual Basic .NET jde ještě dál a explicitní použití ukazatelů vůbec nepodporuje.)

Řízené C++ a datové typy

Datové typy představují efektivní způsob diferencované reprezentace dat a datových struktur, které reflektují určitý typ informací. Každý programovací jazyk obsahuje vestavěnou množinu datových typů, jejichž pomocí mohou programátoři popisovat datové struktury, charakterizovat jejich pracovní náplň a definovat styl zpracování informací uložených v těchto strukturách. Interní kompozice programovacích jazyků specifikuje takzvané primitivní datové typy, které kompilátor daného jazyka bezpečně zná a je tudíž schopen s nimi bez jakýchkoliv potíží pracovat. Architektura většiny programovacích jazyků je však otevřená, což znamená, že kromě primitivních typů je možné pracovat také s jinými, uživatelsky definovanými datovými typy. Jelikož kompilátor dotyčného programovacího jazyka tyto typy nezná (na rozdíl od typů vestavěných), musejí být před svým použitím definovány podle syntaktických pravidel použitého jazyka. Pokud budeme abstrahovat od jiných v současné době populárních programovacích jazyků, mezi které patří Visual Basic, C# nebo Java, a zaměříme se výsostně na C++, můžeme mluvit o několika základních uživatelsky definovaných typech, jakými jsou třídy, struktury, uniony či enumerační (výčtové) typy. Nativní C++ bylo v otázce datových typů značně flexibilní, protože vám nabízelo možnosti nejenom pro vytváření těchto typů, ale také pro určení způsobu jejich bitové interpretace v paměti počítače.

Rozhodnete-li se přejít na řízené C++, budete muset respektovat zákonitosti a pravidla, která jsou tomuto novému vývojově-exekučnímu prostředí vlastní. Tato pravidla mimo jiné definují styl práce s datovými typy. Předně byste si měli osvojit novou koncepci klasifikace datových typů. V prostředí jazyka C++ s Managed Extensions můžeme typy rozdělit na hodnotové a odkazové (referenční).

POZNÁMKA



V řízeném C++ se hodnotové datové typy označují také jako `__value` typy a odkazové datové typy zase jako `__gc` typy. Tato symbolika je odvozena od klíčových slov, které se využívají při definici příslušných skupin typů v jazyce C++ s Managed Extensions.

Společný typový systém platformy .NET Framework předepisuje, že tak hodnotové jakožto i odkazové datové typy budou mít své přímé zastupitele v systémové vrstvě. Těmto zastupitelům se říká systémové datové typy.

Hodnotové datové typy

Hodnotové datové typy jsou vhodné pro úschovu širokého spektra informačních hodnot, mezi které patří především celočíselné a reálné hodnoty s různým definičním oborem, dále logické hodnoty `true` a `false` nebo textové znaky. Přehled hodnotových datových typů, které jsou kompatibilní se společnou jazykovou specifikací, zobrazuje tab. 1.3.

Tab. 1.3: Přehled CLS-kompatibilních hodnotových datových typů

Systémový hodnotový datový typ	Obor hodnot	Charakteristika
`System::Boolean`	Logické hodnoty `true` a `false`	Datový typ `System::Boolean` je schopen reprezentovat logické hodnoty `true` a `false`. Tyto hodnoty je možné použít například v rozhodovacích příkazech či cyklech a ovlivnit tak další běh programu.
`System::Byte`	8bitové celočíselné hodnoty bez znaménka z intervalu <0, 255>	Datový typ `System::Byte` dokáže pracovat s 256 celočíselnými hodnotami včetně nuly. Pro svou kapacitní nenáročnost (proměnná tohoto typu alokuje jenom 1 bajt), je vhodným kandidátem na typ pro řídící proměnné cyklů či jiné datově nenáročné entity.
`System::Char`	Znaky znakové sady Unicode	Proměnné datového typu `System::Char` vědí pracovat se znaky sady Unicode. Interně jsou hodnoty tohoto typu představovány 16bitovými celočíselnými hodnotami z intervalu <0, 65535>.
`System::DateTime`	Časové a datumové hodnoty	Budete-li potřebovat datový typ, jehož pomocí lze uskutečňovat operace s časovými a datovými hodnotami, typ `System::DateTime` je vám k službám. Typ pokrývá časové hodnoty od půlnoci (0:00:00) až do konce dne (23:59:59) a datové hodnoty od 1. ledna roku 1 po 31. prosince roku 9999.
`System::Int16`	16bitové celočíselné hodnoty se znaménkem z intervalu <-32768, 32767>	Datový typ `System::Int16` je základním typem pro práci se 16bitovými celočíselnými hodnotami. Alokační kapacita proměnné tohoto typu je 2 bajty.

`System::Int32`	32bitové celočíselné hodnoty se znaménkem z intervalu <-2147483648, 2147483647>	Na 32bitových počítačových platformách, které pozůstávají z 32bitových procesorů a 32bitových operačních systémů jde o nejefektivnější typ pro realizaci operací se 4bajtovými integrálními hodnotami.
`System::Int64`	64bitové celočíselné hodnoty se znaménkem z intervalu <-9223372036854775808, 9223372036854775807>	`System::Int64` je celočíselným datovým typem s nejširším oborem hodnot. Jelikož jsou jeho instance kapacitně náročnější (každá spolkne 8 bajtů), je vhodné tento typ používat pouze při opodstatněných příležitostech.
`System::Single`	32bitová reálná čísla se znaménkem s jednoduchou přesností z intervalu <-3.402823e38, 3.402823e38>	Proměnné datového typu `System::Single` jsou vhodné pro zpracovávání vědeckých výpočtů s 32bitovými reálnými čísly, pro které je dostačující jednoduchá přesnost výpočtu.
`System::Double`	64bitová reálná čísla se znaménkem s dvojitou přesností z intervalu <-1.79769313486232e308, 1.79769313486232e308>	Na rozdíl od proměnných datového typu `System::Single` není pro proměnné typu `System::Double` problémem pracovat s 64bitovými reálnými čísly s dvojitou přesností. Tento datový typ je vhodný zejména pro náročné matematické propočty, v nichž je nutno eliminovat odchylky způsobené použitím jednoduché přesnosti výpočtu.
`System::Decimal`	128bitová reálná čísla	Obor hodnot datového typu `System::Decimal` je velice široký a bez jakéhokoliv přehánění můžeme říci, že daleko přesahuje možnosti běžných programových operací. Vzhledem k tomu, že jde o kapacitně nejnáročnější datový typ (1 proměnná alokuje 16 bajtů), měli byste jeho instance zapájet do práce pouze ve skutečně odůvodněných případech.

Všechny hodnotové datové typy, jež jsou uvedeny v tab. 1.3, jsou odvozeny od třídy `System::ValueType`. Třída `System::ValueType` je implicitní podtřídou primární bázové třídy `System::Object`, ostatně podobně jsou na tom také všechny další řízené třídy bázové knihovny tříd. Je důležité, abyste si uvědomili, že výše popsané hodnotové datové typy disponují plnou konformitou k standardům společné jazykové specifikace CLS. To tedy znamená, že tyto datové typy, respektive jejich instance, je možné použít z libovolného .NET-kompatibilního programovacího jazyka. Pokud byste se rádi v budoucnu věnovali aplikování koncepce programování ve více programovacích jazycích platformy .NET Framework, anebo byste rádi prakticky vyzkoušeli jazykovou interoperabilitu v řízeném prostředí, měli byste při psaní kódu v jazyce C++ s Managed Extensions používat jenom ty datové typy, které vyhovují požadavkům CLS.

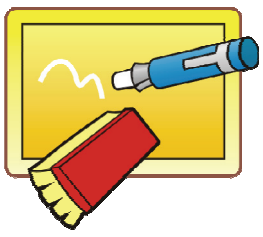
Ať tak či onak, nijak nesmíme přehlédnout i další primitivní hodnotové datové typy, které sice nesplňují standardy společné jazykové specifikace, no přesto je lze v programovém kódu řízeného C++ použít. Jedné se o typy `System::SByte`, `System::UInt16`, `System::UInt32` a `System::UInt64`. Ano, žádný z vyjmenovaných typů není CLS-kompatibilní, to však samozřejmě neznamená, že by tyto typy byly k ničemu. Existují totiž okolnosti a situace, ve kterých i CLS-nekompatibilní datové typy prokazují své silné stránky. Neplánujete-li vytvářet ovládací prvky a komponenty, s nimiž by mohli pracovat i vývojáři jiných .NET-programovacích jazyků, můžete CLS-nekompatibilní typy ve svém zdrojovém kódu používat. Pokud ale bude nutné volat váš kód napsaný v řízeném C++ rovněž z Visual Basicu .NET nebo z C#, pak uděláte dobře, když se CLS-nekompatibilním datovým typům raději vyhnete. Bližší charakteristiku třech neznaménkových a jednoho znaménkového typu můžete nalézt v tab. 1.4.

Tab. 1.4: Popis CLS-nekompatibilních hodnotových datových typů System::SByte, System::UInt16, System::UInt32 a System::UInt64

Systémový hodnotový datový typ	Obor hodnot	Charakteristika
`System::SByte`	8bitová celá čísla se znaménkem z intervalu <-128, 127>	Datový typ `System::Sbyte` je, co se oboru hodnot týče, nejméně vyspělým datovým typem. Jeho instance si poradí jenom s 256 hodnotami ze stanoveného intervalu. Výhodou proměnných tohoto typu je opravdu malá kapacitní náročnost (1 bajt). Alternativním CLS-kompatibilním datovým typem je `System::Int16`.
`System::UInt16`	16bitové celočíselné hodnoty bez znaménka z intervalu <0, 65535>	Definiční obor datového typu `System::UInt16` tvoří nula a 65535 kladných přirozených čísel. V případě, že si nepřejete pracovat s CLS-nekompatibilním datovým typem, můžete místo typu `System::UInt16` použít typ `System::Int32` (za jistých okolností rovněž typ `System::Int16`).
`System::UInt32`	32bitové celočíselné hodnoty bez znaménka z intervalu <0, 4294967295>	Datový typ `System::UInt32` nachází své uplatnění při práci se středně objemnými 32bitovými datovými hodnotami bez znaménka. Není-li pro vás rozhodující alokační kapacita, můžete tento typ substituovat typem `System::Int64`, jenž navíc plně vyhovuje pravidlům společné jazykové specifikace.
`System::UInt64`	64bitové celočíselné hodnoty bez znaménka z intervalu <0, 18446744073709551615>	Datový typ `System::UInt64` je zpomezi všech neznaménkových datových typů největším přeborníkem – poradí si totiž s největšími celočíselnými hodnotami. Toto pozitivum je však vyváženo vyšší kapacitní náročností: jedna proměnná typu `System::UInt64` alokuje 8 paměťových bajtů. Je-li vašemu srdci bližší CLS-kompatibilní typ, zvolte `System::Decimal`, ovšem mějte na paměti, že jeho alokační kapacita se šplhá k 16 bajtům na proměnnou.

Pracujete-li s podporou řízených rozšíření jazyka C++, kompilátor bude výskyt datových typů standardního C++ nahrazovat jejich řízenými ekvivalenty.

POZNÁMKA



Podpora řízených rozšíření jazyka C++ se aktivuje pomocí přepínače kompilátoru `/clr`. Pokud je specifikován tento přepínač, výstupem kompilátoru produktu Visual C++ .NET 2003 bude plnohodnotné sestavení aplikace .NET obsahující moduly s MSIL kódem, metadata a aplikační manifest. Založíte-li aplikaci .NET s využitím projektových šablon, o nichž jsme se zmiňovali výše, pak bude přepínač `/clr` aktivován automaticky. Zapnutí uvedeného přepínače způsobí, že všechny funkce, které napíšete, budou řízené. To však neplatí pro uživatelsky definované datové typy, jakými jsou třídy, struktury a rozhraní: při definici řízených variant těchto typů musíte výslovně uvést klíčové slovo `__gc` (zamýšlíte-li definovat odkazový typ), nebo

	klíčové slovo `__value` (v případě definice hodnotového typu). (Jelikož rozhraní je odkazovým typem, do úvahy přichází pouze definice pomocí klíčového slova `__gc`.) Programovací jazyk C++ s Managed Extensions vám však umožňuje i v řízeném kódu definovat neřízenou funkci, tedy funkci, jejíž exekuce nebude pod správou běhového prostředí CLR. V okamžiku, kdy bude doručen požadavek na zpracování takovéto funkce, CLR svěří jej exekuci nativní platformě. Výskyt neřízené funkce v řízeném programovém kódu musí být zřetelně vymezen direktivou `#pragma unmanaged`.

To znamená, že když budete v řízeném kódu deklarovat proměnnou typu `bool`, kompilátor změní typ proměnné na systémový typ `System::Boolean`. Podobně se chovají také další hodnotové typy klasického C++ (tab. 1.5).

Tab. 1.5: Přehled řízených hodnotových typů a jejich C++ ekvivalentů

Řízený hodnotový typ	C++ ekvivalent
`System::Boolean`	`bool`
`System::Byte`	`unsigned char`
`System::SByte`	`char, signed char`
`System::Char`	`wchar_t`
`System::DateTime`	`Žádný ekvivalent`
`System::Int16`	`short`
`System::UInt16`	`unsigned short`
`System::Int32`	`int, long`
`System::UInt32`	`unsigned int, unsigned long`
`System::Int64`	`__int64`
`System::UInt64`	`unsigned __int64`
`System::Single`	`float`
`System::Double`	`double`
`System::Decimal`	`Žádný ekvivalent`

Jak si můžete všimnout, existují tři řízené hodnotové datové typy, které substituují více ekvivalentních datových typů standardního C++. Jde o typy `System::Sbyte`, `System::Int32` a `System::UInt32` (CLS-kompatibilním typem je ovšem pouze typ `System::Int32`). Jestli v jazyce C++ s Managed Extensions použijete při deklaraci proměnných klíčová slova `char` a `signed char`, kompilátor interně oba typy nahradí typem `System::SByte`. Podobná je situace také u typů `int` a `long` (tyhle budou nahrazeny typem `System::Int32`), a rovněž u typů `unsigned int` a `unsigned long` (za tyto typy bude dosazen typ `System::UInt32`). Když vezmeme v potaz tato pravidla, mohli bychom nabýt dojmu, že nelze napsat kolekci dvou přetížených metod, z nichž jedna by disponovala formálním parametrem typu `int` a další formálním parametrem typu `long`. Dobrou zprávou je, že kompilátor vás v tomto směru nijak neomezuje, neboť signaturu metody s parametrem typu `long` automaticky doplní o modifikátor `Microsoft.VisualC.IsLongModifier` (v jazyce MSIL má datový typ generovaného parametru podobu `int32 modopt ([Microsoft.VisualC] Microsoft. VisualC. IsLongModifier`.

Formální parametr první metody, jenž je typu `int`, bude v jazyce MSIL nahrazen typem `int32`, čímž jsme dospěli k tomu, že je vždy možné jednoznačně identifikovat správnou verzi přetížené metody. Stejně se kompilátor chová při pokusu o přetížení metod použitím typů `char` a `signed char` (na parametr typu `char` bude aplikován modifikátor `Microsoft.VisualC.NoSignSpecifiedModifier`) či typů `unsigned int` a `unsigned long` (na parametr typu `unsigned long` bude aplikován modifikátor `Microsoft.VisualC.IsLongModifier`).

V zásadě můžeme prohlásit, že je pouze na programátorovi, zda se rozhodne při deklaraci proměnných hodnotových datových typů používat klíčová slova reprezentující systémové datové typy (jako třeba `System::Byte` nebo `System::Int64`), anebo dá přednost klíčovým slovům nativního C++ a jejich mapování na řízené typy přenechá kompilátoru. Pokud smíme, doporučovali bychom vám raději používat systémové hodnotové datové typy, protože již z pohledu na ně je každému okamžitě jasné, jaké datové typy jste zamýšleli použít ve svých fragmentech zdrojového kódu. Vyjma toho, použití systémových typů je velice užitečné také při programování ve více .NET-programovacích jazycích (jisté datové typy nativního C++ nemusejí jiné jazyky obsahovat, no pokud použijete systémový typ, společná jazyková specifikace a společný typový systém garantují, že tento typ budete moci bez jakýchkoliv potíží použít i ve Visual Basicu .NET nebo v C#).

Enumerační (výčtové) datové typy

Enumerační typy představují uživatelsky definované hodnotové datové typy, které je možné použít pro ukládání konstantních hodnot jisté kolekce členů. Členy enumerace jsou známy také jako enumerátory – ve skutečnosti jde o pojmenované konstanty, které tvoří tělo enumeračního typu a které může programátor v případě potřeby využít. Enumerační typy, jež se vyskytují v nativním C++, jsou deklarovány pomocí klíčového slova `enum`:

```
enum Barvy
{
    Cervena = 10,
    Zelena = 20,
    Modra = 30
};
```

Uvedený fragment programového kódu definuje enumerační typ s názvem `Barvy`, který obsahuje tři členy, nebo enumerátory, chcete-li. Názvy těchto členů jsou `Cervena`, `Zelena` a `Modra`. Enumerátory jsou konstanty, přičemž však není rozhodující, zda jejich inicializaci uskuteční programátor, anebo tuto činnost svěří do rukou kompilátoru. V případě enumerace `Barvy` jsou všechny enumerátory explicitně inicializovány celočíselnými hodnotami. Pokud by programátor členy enumeračního typu neinicializoval, nedošlo by k žádné chybě, protože jak jsme si již řekli, za těchto okolností by enumerátory inicializoval kompilátor prostřednictvím implicitních hodnot. Tato implicitní inicializace by byla vykonána podle matematických pravidel aritmetické posloupnosti s diferencí 1 a počátkem v nulovém bodě. Řečeno méně matematicky: prvnímu enumerátoru by byla přiřazena hodnota 0, druhému 1, třetímu 2 atd. Jsou-li enumerátory inicializovány kompilátorem, pak ve všeobecnosti platí, že hodnota následujícího enumerátoru je o jednotku větší než hodnota předcházejícího enumerátoru. Samozřejmě je možné a za určitých okolností dokonce i velice prospěšné kombinovat oba druhy inicializace enumerátorů. Podle tohoto postupu může některé členy enumeračního typu inicializovat programátor, zatímco ostatní budou inicializovány za pomoci kompilátoru:

```
enum Barvy
{
   Cervena,
   Zelena = 20,
   Modra,
   Oranzova = 15,
   Zluta
};
```

Jednotlivé enumerátory mají nyní tyto inicializační hodnoty: `Cervena = 0`, `Zelena = 20`, `Modra = 21`, `Oranzova = 15`, `Zluta = 16`. Hodnotu enumerátoru lze umístit do externí celočíselné proměnné například takhle:

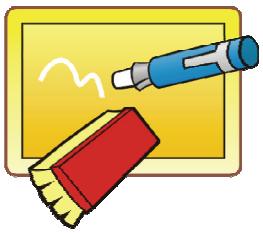
```
int Barva = Barvy::Zluta;
```

Až doposavad jsme mluvili o enumeračních typech nativního C++. V jazyce C++ s Managed Extensions patří enumerace mezi hodnotové datové typy a tato skutečnost si také vyžádala zásah do způsobu jejich definice. Enumerační typy se proto v řízeném prostředí definují pomocí spojení klíčových slov `__value enum`, čímž jsou programátoři ihned upozorněni na to, že byla zahájena definice hodnotového enumeračního typu. Nicméně, vsunutí klíčového slova `__value` před slovo `enum` je z vizuálního hlediska jediná podstatní změna, která se týká definičního příkazu. Způsob inicializace zůstává i v řízeném C++ neměnný. Pokud bychom měli z už představeného enumeračního typu `Barvy` udělat hodnotový enumerační typ, který bychom mohli použít v řízeném C++, výsledek by měl tuto podobu:

```
__value enum Barvy
{
   Cervena = 10,
   Zelena = 20,
   Modra = 30
};
```

V řízeném kódu byste měli používat pouze hodnotové enumerační typy definované jako `__value enum`. Přesněji řečeno, jakýkoliv enumerační typ, jenž bude definován v tělech hodnotových struktur (`__value struct`) či odkazových struktur (`__gc struct`), anebo v tělech hodnotových tříd (`__value class`) či odkazových tříd (`__gc class`), by měl být definován prostřednictvím klíčových slov `__value enum`.

POZNÁMKA



Definice klasického enumeračního typu jazyka C++, která se nachází v těle řízeného datového typu, vyvolá chybovou výjimku, na kterou vás kompilátor upozorní. Problém vyřešíte tak, že nativní enumerační typ převedete do jeho řízené podoby za asistence klíčových slov `__value enum`.

Instance hodnotových enumeračních typů jazyka C++ s Managed Extensions jsou odvozeny od třídy `System::Enum` (tato je zase odvozena od třídy `System::ValueType`). Hodnotový enumerační typ vždy pracuje s bázovým datovým typem, jímž je ve většině případů typ `int` čili `System::Int32`. Bázový datový typ enumeračního typu deklaruje charakter jednotlivých enumerátorů, což znamená, že když neuvedete jiný, bude implicitně zvolen bázový typ `System::Int32`. Hodnotové enumerační typy se od svých předchůdců z klasického C++ liší v tom, že v řízeném C++ je možné přímo determinovat

bázový typ enumeračního typu, zatímco v nativním C++ je situace jiná. Budete-li chtít definovat enumeraci, jejímž bázovým datovým typem bude `char`, postupujte následovně:

```
__value enum DnyVTydnu : char
{
   Pondeli, Utery, Streda, Ctvrtek, Patek,
   Sobota, Nedele
};
```

Na první pohled to možná vypadá, jako kdyby byl enumerační typ `DnyVTydnu` odvozen od typu `char`, ovšem to je pochopitelně pouhé zdání. Operátor dvojtečka (`:`) v tomto zápisu říká, že enumerátory definovaného enumeračního typu mohou uchovávat celočíselné hodnoty, jež jsou platné pro datový typ `char`. Jestliže by některý z enumerátorů obsahoval hodnotu, která by přesahovala povolený celočíselný interval typu `char` (<-128, 127>), kompilátor jazyka C++ s Managed Extensions by vygeneroval varování druhé úrovně s číselní identifikací C4309. K nastíněné situaci dojde kupříkladu v níže uvedeném zdrojovém kódu:

```
__value enum DnyVTydnu : char
{
   Pondeli, Utery, Streda, Ctvrtek, Patek,
   Sobota, Nedele = 200
};
```

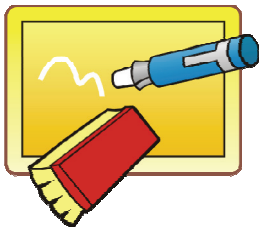
Všimněte si, že do enumerátoru `Nedele` jsme uložili hodnotu 200, která evidentně překračuje meze datového typu `char`. Kompilátor chybu odhalí, přičemž zobrazí varování, v němž nás obeznámí se skutečností, že hodnota daného členu enumerace bude ořezána. Použijeme-li následující kód a aplikaci spustíme, v okně se zprávou se objeví hodnota -56.

```
char x = DnyVTydnu::Nedele;
MessageBox::Show(x.ToString());
```

Kromě toho, že hodnoty enumerátorů můžeme přiřazovat do proměnných adekvátních datových typů, je možné uplatnit i opačný přístup – odpovídající celočíselní hodnotu můžeme uložit také do instance enumeračního typu. Zde se však nevyhneme povinné realizaci konverzních operací. Věnujte pozornost dalšímu výpisu zdrojového kódu:

```
DnyVTydnu Den = (DnyVTydnu)4;
System::Object * obj = __box(Den);
MessageBox::Show(obj->ToString());
```

Na prvním řádku deklarujeme proměnnou `Den` enumeračního typu `DnyVTydnu`. Tento datový typ sdružuje sedm členů, jež jsou implicitně naplněny hodnotami 0 až 6. My však chceme do vytvořené instance enumeračního typu uložit hodnotu 4. To sice můžeme provést, no musíme ji správně přetypovat na typ cílové enumerace. Vzhledem k tomu, že si přejeme získat textovou reprezentaci hodnoty proměnné `Den`, využíváme mechanizmus sjednocení typů (angl. boxing), díky kterému zabezpečíme unifikaci zúčastněných datových typů (hodnotový typ bude konvertovaný na objekt odkazového typu, na který bude nasměrován řízený (`__gc*`) ukazatel uložený v odkazové proměnné `obj`).

POZNÁMKA	
	O mechanizmu sjednocení typů, jakožto i o jeho zpětném chodu (angl. unboxing) si budeme povídat až v dalších kapitolách této vývojářské příručky. V tuto chvíli vám bude stačit, když si povíme, že mechanizmus sjednocení typů provádí obousměrné konverzní operace mezi instancemi hodnotových a odkazových datových typů. Na rozdíl od některých jiných .NET-programovacích jazyků, například Visual Basicu či C#, není mechanizmus sjednocení typů v řízeném C++ realizován implicitně. Jinými slovy, vývojář musí uskutečnit všechny operace, které jsou s aktivací a exekucí mechanizmu sjednocení typů spojeny. Důvodem takovéhoto počínání je mimo jiné také skutečnost, že konverzní operace jsou vždy zdrojem potenciálních výkonnostních penalizací, což jsou nepříznivé interference, které se snažíme v maximální možné míře eliminovat.

Je-li instance enumeračního typu zabalena, můžeme ji použít k volání metody `ToString`, která nám ochotně poskytne žádanou textovou informaci. V dialogu bude zobrazen textový řetězec „Patek“, což je v pořádku, protože hodnota 4 přináleží právě pátému členu enumerace `DnyVTydnu`.

Hodnotové struktury a hodnotové třídy

Tak hodnotové struktury jakožto i hodnotové třídy patří do skupiny hodnotových datových typů, což s sebou nese několik důsledků. Zaprvé, nominální alokační kapacita těchto instancí není nijak veliká, z čehož vyplývá, že tyto datové typy jsou vhodné pro práci s nepříliš náročnými kolekcemi dat. Zadruhé, hodnotové typy nejsou ukládány na řízenou hromadu, takže postranní režijní náklady spojené s jejich alokací a dealokací nejsou závratně vysoké. Instance hodnotových struktur a tříd nacházejí své místo na zásobníku programového vlákna, kde sídlí společně s objekty jiných hodnotových typů.

UPOZORNĚNÍ	
	V případě, kdy se definice hodnotové struktury nebo hodnotové třídy nachází v těle odkazové struktury nebo odkazové třídy, může být instance hodnotové struktury, respektive třídy uložena (společně s instancí nadřazeného odkazového typu) na řízené hromadě a nikoliv na zásobníku vlákna.

Hodnotové struktury a třídy se definují pomocí klíčových slov `__value struct` a `__value class`. V prostředí programovacího jazyka C++ s Managed Extensions neexistují zásadní sémantické rozdíly mezi strukturami a třídami. (Pokusme se tuto skutečnost porovnat například s jazyky Visual Basic a C#, kde jsou pojmy struktury a třídy velice rozdílné: Struktury jsou vždy zástupci hodnotových typů, zatímco třídy naopak vždy představují odkazové datové typy.) Jestli tedy programujete v řízeném C++, směrodatné pro vás bude, zda pracujete s hodnotovými nebo odkazovými strukturami a třídami. Odkazové struktury a třídy se definují prostřednictvím klíčových slov `__gc struct` a `__gc class`, přičemž jejich instance jsou, jak již víte, umísťovány na řízenou hromadu. Podrobnější exkurzi do světa těchto odkazových typů si však ponecháme až na později.

Podívejme se nyní na prostou hodnotovou strukturu s názvem `Formular`:

```
#include "StdAfx.h"
#using <mscorlib.dll>

__value struct Formular
{
   Formular()
   {
      System::Windows::Forms::Form __gc * frm =
                  new System::Windows::Forms::Form;
      frm->Text = S"Formulář z Windows Forms.";
      frm->Show();
   }
};
```

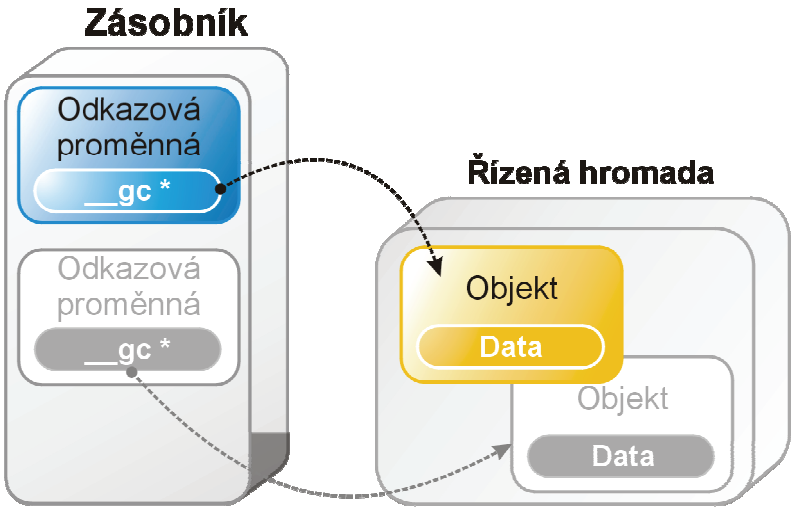
Hodnotová struktura `Formular` obsahuje jeden bezparametrický konstruktor, v jehož těle dochází k instanciaci třídy `Form` z jmenného prostoru `System::Windows::Forms`. Ačkoliv jsme se ještě podrobně nezabývali vytvářením řízených objektů, zcela jistě nezaškodí, když si malou ukázku předvedeme již nyní. Zrod řízených objektů se ve všeobecnosti řídí podle níže popsaných pravidel:

1. Instance řízené třídy, tedy řízený objekt, je společným běhovým prostředím CLR platformy .NET Framework uložen do vyhrazené oblasti operační paměti počítače, které se říká řízená hromada.

2. Na řízený objekt směruje řízený (`__gc*`) ukazatel, jehož pomocí je dotyčný objekt z prostředí aplikace .NET dosažitelný. Řízený ukazatel se od standardního ukazatele liší tím, že je typově bezpečný a může být namířen jedině na platné instance řízených programových entit.

3. Řízený ukazatel sleduje automatická správa paměti, která kontroluje objekty uskladněné na řízené hromadě. Jakmile nebude objekt z programového kódu aplikace .NET dosažitelný, automatický správce paměti jej z řízené hromady uvolní. (Proces likvidace řízených objektů z paměti je však mnohem komplikovanější, kupříkladu také v závislosti na tom, zda budou před samotnou destrukcí objektů volány rovněž jejich finalizační metody či nikoliv.)

Grafickou ilustraci vztahu řízených (`__gc*`) ukazatelů a instancí odkazových (`__gc`) tříd můžete vidět na obr. 1.1.



Obr. 1.1: Interakce mezi řízenými ukazateli a řízenými objekty

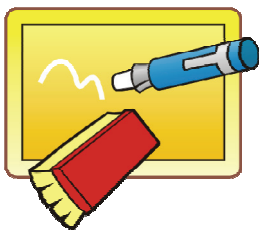
Hodnotové struktury a třídy jsou odvozeny od třídy `System::ValueType`, která zase dědí své charakteristiky od primární bázové třídy `System::Object`. Jelikož existuje vzájemná vazba mezi hodnotovými typy a třídou `System::ValueType`, je možné, aby hodnotové struktury nebo třídy překrývaly metody této bázové třídy. Pokud bychom například chtěli překrýt metodu `ToString` třídy `System::ValueType`, mohli bychom použít tento postup:

```
__value struct A
{
    String * ToString()
    {
        return S"Struktura A";
    }
};
```

Použití hodnotových struktur a tříd je v podstatě shodné, no na paměti byste měli mít skutečnost, že obě uvedené entity nabízejí rozdílnou úroveň přístupu ke svým datovým členům. Budete-li v těle hodnotové struktury definovat funkci, tato bude implicitně veřejně přístupná, a tedy bude viditelná i pro vnější klientský programový kód. Na druhou stranu, bude-li ta samá funkce definovaná v těle hodnotové třídy, bude implicitně privátní, což znamená, že žádný kód, kromě kódu, jenž se nachází v těle této třídy, nebude mít k definované funkci přístup. Viditelnost jednotlivých datových členů hodnotových struktur a tříd ovšem můžete upravit použitím vhodných modifikátorů přístupu.

Hodnotové struktury a třídy jsou implicitně zapečetěné, důsledkem čehož je, že nemohou působit jako bázové entity, od kterých by byly odvozeny další hodnotové struktury nebo třídy.

POZNÁMKA



Jazyk C++ s Managed Extensions zavádí nové klíčové slovo `__sealed`, které pokud je vloženo před definici podporované entity, způsobí, že tato entita bude zapečetěná. Aplikace klíčového slova `__sealed` není při hodnotových strukturách a třídách nutná, neboť tyto jsou již „od výroby“ definované jako zapečetěné. Navzdory tomu kompilátor akceptuje výskyt klíčového slova `__sealed` v rámci definice hodnotových typů, a proto můžeme prohlásit, že následující struktura je definována správně:

```
// Explicitní uvedení klíčového slova __sealed v definici
// struktury.
__sealed __value struct A
{
    // Kód těla struktury je vynechán...
};
```

Hodnotové struktury a třídy nemohou být odvozeny od žádného jiného hodnotového nebo odkazového datového typu. Smí však implementovat libovolný počet řízených (`__gc`) rozhraní. Když se hodnotový typ rozhodne implementovat jisté rozhraní, musí uskutečnit definici všech členů příslušného rozhraní.

Jak jste se před chvíli přesvědčili, v těle hodnotového datového typu se může vyskytovat také řízený ukazatel na objekt řízené třídy (viz výše uvedená struktura `Formular`). Za těchto podmínek je instance hodnotového typu uložená na zásobníku vlákna a objekt, na který směruje řízený ukazatel, se nachází na řízené hromadě. Jestli ale hodnotový typ

neobsahuje `__gc*` ukazatel na objekt řízené třídy, může být jeho instance vytvořena rovněž na standardní, tedy neřízené (nativní) hromadě pomocí operátoru `__nogc new`. V tomto případě je však potřebné instanci hodnotového datového typu z neřízené hromady ve vhodném okamžiku explicitně odstranit, protože jinak by došlo ke vzniku „paměťové díry", tedy docela závažné a poměrně těžko identifikovatelné chybě.

Charakteristika odkazového datového typu System::String

Programovací jazyk C++ s Managed Extensions předkládá před programátory a vývojáře robustní aparát pro práci s textovými řetězci. Bázová knihovna tříd vývojově-exekuční platformy .NET Framework 1.1 definuje řízenou třídu `String`, která je umístěna v kořenovém jmenném prostoru `System`. Instance této třídy mohou uchovávat řetězce textových znaků sady Unicode. Po pravdě řečeno, při práci v řízeném C++ se budete setkávat jenom se znakovou sadou Unicode, a tudíž nemusíte mít žádné obavy o dostupnou paletu textových znaků, a to jak regulárních, tak doslovných. Každý textový řetězec, který vytvoříte, je uložen do předem připravené instance třídy `System::String`. Kdybychom se pokusili prozkoumat textový řetězec v prostředí .NET pod drobnohledem, zjistili bychom, že tato entita je ve skutečnosti kolekcí objektů třídy `System::Char`. Je tedy zřejmé, že mezi instancemi tříd `System::Char` a `System::String` je velice úzký vzájemný vztah.

Jelikož je každý textový řetězec uchovávaný v instanci třídy `System::String`, platí pro proces zakládání instancí tohoto typu všechna pravidla, s nimiž se můžete setkat při jakémkoliv jiném odkazovém datovém typu. To znamená, že instance třídy `System::String` je v průběhu svého vytvoření umístěna do nulté generace řízené hromady a vzápětí je inicializována kolekcí instancí třídy `System::Char`, jež formují požadovaný textový řetězec. Odkaz na vytvořenou instanci třídy `System::String` je uložen do připravené odkazové proměnné typu `System::String*` (`__gc` ukazatel na řízenou instanci třídy `System::String`). Jazyk C++ s Managed Extensions umožňuje programátorům aplikovat mnoho variant instanciace třídy `System::String`. Některé z nich přibližuje následující výpis programového kódu.

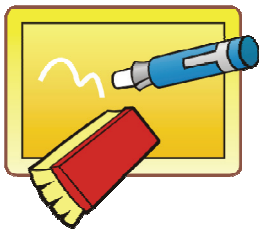
```
// Přímé založení instance třídy System::String.
System::String * str1 = S"Textový řetězec";

// Založení instance třídy System::String pomocí operátoru new.
System::String __gc * str2 = new System::String(S"Textový řetězec");

// Instanciace třídy System::String za asistence přetíženého konstruktoru,
// jemuž je poskytnut ukazatel na textový znak datového typu __wchar_t.

__wchar_t wch1 = 'A';
const __wchar_t * cp_wch1 = &wch1;
System::String __gc * str3 = __gc new System::String(cp_wch1);
```

V prvním případě je instance třídy `System::String` vytvořená přímo: Do proměnné `str1` typu `System::String*` je uložen odkaz na instanci třídy `System::String`, která obsahuje textový řetězec specifikovaný nalevo od operátoru přiřazení. Všechny textové řetězce musejí být zapsány v dvojitých uvozovkách. Tento požadavek vás určitě nepřekvapí, avšak novinkou může být použití textového prefixu `S`. Nachází-li se před textovým řetězcem tento prefix, je generovaný řízený textový řetězec. Ve skutečnosti byste měli ve spojení s textovými řetězci používat prefix `S` vždy, nakolik realizace operací s řízenými řetězci je na platformě .NET efektivnější.

Druhý instanciační příkaz upotřebuje operátor `new`, přičemž explicitně předává konstruktoru třídy `System::String` řízený textový řetězec. Všimněte si, že typem odkazové proměnné `str2` je `System::String __gc*`, který výslovně poukazuje na skutečnost, že pracujeme s řízeným ukazatelem nasměrovaným na instanci řízeného odkazového datového typu.

Konstruktor třídy `System::String` disponuje několika přetíženými variantami. Třetí způsob instanciace třídy `System::String` pracuje s jednou verzí přetíženého konstruktoru, která v podobě argumentu přijímá ukazatel na hodnotu typu `const char` (modifikátor `const` znemožňuje změnu hodnoty typu `__wchar_t`, na kterou je ukazatel nasměrován, čili například po zadání příkazu `*cp_wch1 = 'B';` by kompilátor jazyka C++ s Managed Extensions vygeneroval chybové hlášení). Tato ukázka je zajímavá také z jiného pohledu, protože explicitně využívá operátoru `__gc new`, jenž vizuálně naznačuje, že na tomto místě dochází k založení nové instance řízené třídy.

POZNÁMKA



Pokud je do aplikačního projektu začleněn odkaz na jmenný prostor `System`, což je standardní nastavení, můžete název tohoto jmenného prostoru při deklaraci odkazových proměnných typu `System::String*` vynechat. Pak budete pracovat s typem `String*`.

Realizace operací s textovými řetězci

V okamžiku, kdy je instance třídy `System::String` správně vytvořena, můžeme volat její vlastnosti a metody. Níže je uvedená hrstka praktických programových ukázek, jež demonstrují flexibilitu instancí třídy `System::String`.

Ukázka 1: Získání libovolného znaku v textovém řetězci

Textový řetězec je ve své podstatě jistou posloupností textových znaků, a proto by neměl být problém získat reprezentaci kteréhokoliv znaku z textového řetězce. S touto úlohou nám pomůže vlastnost `Chars` instance třídy `System::String`. K bloku `get_` této vlastnosti se dostaneme před metodu `get_Chars`. Nabídneme-li metodě `get_Chars` index, respektive pozici textového znaku v řetězci, který si přejeme získat, metoda nám vybraný znak odevzdá v podobě hodnoty typu `__wchar_t`. Pozice znaků v textovém řetězci se počítají od nuly, což znamená, že první znak disponuje indexem 0, druhý indexem 1, třetí indexem 2 atd., přičemž index posledního znaku je o jednotku menší než délka textového řetězce.

```
System::String __gc * str1 = S"Textový řetězec";
// Zobrazení čtvrtého znaku textového řetězce pomocí metody get_Chars.
System::Char Znak = str1->get_Chars(3);
MessageBox::Show(Znak.ToString());
```

TIP	

	Poněvadž je vlastnost `Chars` instance třídy `System::String` indexovaná, není bezpodmínečně nutné, abychom volali metodu `get_Chars`, protože stejný cíl dosáhneme také tehdy, když číselnou pozici textového znaku v řetězci umístíme do hranatých závorek, a tento komplet zapíšeme za název vlastnosti `Chars`:
	
	`// Použití indexované vlastnosti Chars.`
	`System::Char Znak = str1->Chars[3];`
	
	Tímto způsobem můžeme s vlastností `Chars` pracovat jako s polem, elementy kterého vystupují v rolích textových znaků.

Výsledkem činnosti uvedeného fragmentu zdrojového kódu bude zobrazení čtvrtého znaku („t“) v dialogu se zprávou.

Ukázka 2: Použití textového řetězce s nulovou délkou

Textový řetězec s nulovou délkou představuje takzvaný prázdný řetězec, jenž může sloužit například jako indikátor východiskového stavu. Jakoukoliv instanci třídy `System::String` lze inicializovat řetězcem s nulovou délkou pomocí statické vlastnosti `Empty` této třídy:

```
System::String __gc * str1 = S"Textový řetězec";
// Instance třídy System::String, na níž ukazuje odkazová proměnná str1,
// je inicializovaná řetězcem s nulovou délkou.
str1->Empty;
MessageBox::Show(str1);
```

Ukázka 3: Analýza počtu znaků v textovém řetězci

Zjištění počtu znaků v textovém řetězci nepředstavuje až takovou triviální úlohu, jak by se mohlo na první pohled zdát. Možná si myslíte, že byste mohli použít veřejně přístupnou vlastnost `Length` instance třídy `System::String`, no tenhle postup by nemusel být správný. Důvodem je skutečnost, že návratovou hodnotou vlastnosti `Length` je číselní hodnota vyjadřující počet instancí třídy `System::Char`, z nichž je zkoumaný textový řetězec složen. Vlastnost `Length` ovšem neudává informace o počtu znaků sady Unicode, které analyzovaný řetězec doopravdy tvoří. I když je možné vlastnost `Length` použít v mnoha situacích, správnost jejího pracovního postupu nemusí být vždy stoprocentní, nakolik jeden znak sady Unicode může být reprezentován více instancemi třídy `System::Char`. Vzhledem k tomu, že chceme naši analýzu provádět na úrovni jednotlivých textových znaků sady Unicode, použijeme instanci třídy `StringInfo` z jmenného prostoru `System::Globalization`.

```
System::String __gc * str1 = S"Textový řetězec";

System::Globalization::TextElementEnumerator __gc * txtEnumerator =
   System::Globalization::StringInfo::GetTextElementEnumerator(str1);

System::Int32 PocetZnaku = 0;

while(txtEnumerator->MoveNext())
{
```

```
    PocetZnaku++;
}

MessageBox::Show(PocetZnaku.ToString());
```

Postupujeme podle následujícího algoritmu:

1. Deklarujeme odkazovou proměnnou `txtEnumerator`, která bude moci uchovávat referenci na instanci třídy `TextElementEnumerator` z jmenného prostoru `System::Globalization`. Instance této třídy vystupuje v roli enumerátoru, prostřednictvím kterého je možné zkoumat všechny textové znaky řetězce.

2. Aktivujeme statickou metodu `GetTextElementEnumerator` třídy `StringInfo` z jmenného prostoru `System::Globalization`, jíž odevzdáme textový řetězec určený k analýze. Metoda nám vrátí textový enumerátor, nebo přesněji řečeno instanci třídy `TextElementEnumerator`. Odkaz na zmíněnou instanci bude uložen do předem připravené odkazové proměnné `txtEnumerator`.

3. Vytvoříme hodnotovou proměnnou `PocetZnaku` typu `System::Int32` a explicitně ji inicializujeme nulovou hodnotou. Budoucí hodnota této proměnné bude reflektovat počet textových znaků, které byly zpracovány textovým enumerátorem.

4. Definujeme cyklus `while`, jehož iterace budou probíhat tak dlouho, dokud bude instanční metoda `MoveNext` textového enumerátoru (`txtEnumerator`) navracet logickou hodnotu `true`. Metoda `MoveNext` vrací hodnotu `true` vždy, když textový enumerátor ukončil analýzu jednoho textového znaku a přesunul se na další znak. Jestliže textový enumerátor uskuteční průzkum všech dostupných znaků, metoda `MoveNext` vrátí hodnotu `false`.

5. Při každé iteraci cyklu je hodnotová proměnná `PocetZnaku` inkrementována o hodnotu 1.

Zvolený testovací textový řetězec je tvořen kolekcí patnácti textových znaků, a právě tato hodnota bude zobrazena v okně se zprávou.

Ukázka 4: Zřetězení textových řetězců

Při práci s textovými řetězci dochází velice často k situaci, kdy je zapotřebí spojit více textových řetězců do jednoho monolitního textového řetězce. Tento proces sdružování textových řetězců se označuje jako zřetězení. Bohužel, na rozdíl od jazyka C#, v prostředí jazyka C++ s Managed Extensions neexistuje žádná přetížená verze aritmetického operátoru pro sčítání (+), která by byla schopna uskutečňovat zřetězení textových řetězců. Proto musíme sáhnout po statické metodě `Concat` třídy `System::String`, která nám dovoluje spojit několik řetězců do jednoho celku.

```
System::String __gc * str1 = S"Jazyk C++";
System::String __gc * str2 = S" s Managed Extensions";
// Statická metoda Concat třídy System::String si poradí s zřetězením
// většího počtu textových řetězců.
MessageBox::Show(System::String::Concat(str1, str2));
```

Finálním produktem statické metody `String::Concat` bude nový textový řetězec „Jazyk C++ s Managed Extensions“. Metoda existuje ve více přetížených verzích, které vám mohou podat pomocnou ruku při vytváření textových řetězců z rozličných datových

struktur. Níže uvedený fragment programového kódu kupříkladu předvádí flexibilitu statické metody `String::Concat` při spájení řetězců umístěných v řízeném poli.

```
// Instanciace řízeného pole, které je explicitně inicializované
// textovými řetězci.
System::String __gc * pole __gc[] =
   {S"Právě je ", DateTime::Now.Hour.ToString(), S" hodin."};
// Aktivace statické metody Concat třídy System::String uskutečňuje
// zřetězení všech elementů řízeného pole.
MessageBox::Show(System::String::Concat(pole), S"Informace o časomíře",
   MessageBoxButtons::OK, MessageBoxIcon::Information);
```

Interakce s instancemi třídy System::Text::StringBuilder

Každá zrozená instance třídy `System::String` je neměnná, přičemž obsahuje právě ten textový řetězec, který byl do ní uložen v procesu instanciace. Původní textový řetězec proto není možné jakýmkoliv způsobem upravovat či jinak modifikovat. Když použijete některé metody instance třídy `System::String`, jež operují s textovým řetězcem, běhové prostředí CLR zabezpečí vygenerování nové instance třídy `System::String` s upravenou podobou originálního textového řetězce. Jistě uznáte, že tento pracovní model není příliš efektivní, protože při každém pokusu o změnu textového řetězce je vytvořena nová instance třídy `System::String`, do které je uložena nová podoba dotyčného řetězce. Charakterizované omezení instancí třídy `System::String` můžeme eliminovat použitím třídy `StringBuilder` z jmenného prostoru `System::Text`.

```
System::Text::StringBuilder __gc * strb1 =
   new System::Text::StringBuilder(S"První textový řetězec");
strb1->Append(S", druhý textový řetězec.");
MessageBox::Show(strb1->ToString());
```

Instance třídy `System::Text::StringBuilder` disponuje metodami, jejichž pomocí lze modifikovat její obsah. Výše prezentovaný výpis kódu volá metodu `Append`, která realizuje připojení dodatečného textového řetězce k řetězci, jenž již v instanci uložen byl. Rádi bychom poukázali na to, že při této operaci nedochází k tvorbě nové instance třídy `System::Text::StringBuilder` – jednoduše je upraven obsah původně vytvořené instance. Budete-li chtít odstranit z instance třídy `System::Text::StringBuilder` několik textových znaků, můžete aktivovat metodu `Remove`.

```
System::Text::StringBuilder __gc * strb2 =
   new System::Text::StringBuilder(S"Procesor");

// Instanční metoda Remove odstraňuje 4 textové znaky, přičemž
// s odstraňovacím procesem začíná od čtvrtého textového znaku.
strb2->Remove(4, 4);

System::String __gc * Pole __gc[] = {S"Upravený řetězec: ",
strb2->ToString(), Environment::NewLine,
S"Délka řetězce: ", strb2->Length.ToString()};

MessageBox::Show(String::Concat(Pole), S"Informace o textovém řetězci",
   MessageBoxButtons::OK, MessageBoxIcon::Information);
```

Komparace textových řetězců

Vzájemné porovnávání textových řetězců je v jazyce C++ s Managed Extensions o poznání náročnější než v jiných .NET-kompatibilních jazycích. Toto tvrzení je podloženo odlišným stylem práce porovnávacího operátoru (==). Na rozdíl od jazyka C#, kde operátor == uskutečňuje porovnání referencí i obsahů instancí třídy `System::String`, v jazyce C++ s Managed Extensions se porovnávací operátor soustřeďuje pouze na komparaci řízených referencí, jež jsou uloženy v příslušných odkazových proměnných. Jestli tedy dvě odkazové proměnné typu `System::String*` obsahují reference ukazující na různé instance třídy `System::String`, operátor == vrátí logickou hodnotu `false`. Naopak, budou-li reference odkazových proměnných nasměrovány na totožnou instanci, návratovou hodnotou porovnávacího operátoru bude hodnota `true`.

```
__wchar_t PoleZnaku1 __gc[] = {'T','E','X','T','\0'};
__wchar_t PoleZnaku2 __gc[] = {'T','E','X','T','\0'};


System::String __gc * str1 =
   new System::String(PoleZnaku1);

System::String __gc * str2 =
   new System::String(PoleZnaku2);

if (str2 == str1)
    MessageBox::Show(S"Reference jsou shodné.");
else
    MessageBox::Show(S"Reference nejsou shodné.");
```

Po zpracování tohoto kódu budou zobrazeny informace o neshodnosti referencí. Jak vidíte, operátor == testuje výlučně reference na instance třídy `System::String` a nikoliv samotný obsah těchto instancí. Při požadavku na realizaci testu shodnosti obsahu instancí můžeme aktivovat statickou metodu `Equals` třídy `System::String`, která porovnává obsahy instancí třídy `System::String` na základě velikosti textových znaků (jedná se tedy o „case-sensitive" porovnávací test).

```
__wchar_t PoleZnaku1 __gc[] = {'T','E','X','T','\0'};
__wchar_t PoleZnaku2 __gc[] = {'T','E','X','T','\0'};

System::String __gc * str1 =
   new System::String(PoleZnaku1);

System::String __gc * str2 =
   new System::String(PoleZnaku2);

if (System::String::Equals(str2,str1))
    MessageBox::Show(S"Textové řetězce jsou shodné.");
else
    MessageBox::Show(S"Textové řetězce nejsou shodné.");
```

Statická metoda `System::String::Equals` nám pomůže zjistit, zda obě instance třídy `System::String` disponují stejným obsahem, a sice textovým řetězcem „Text", jenž vznikl agregací elementů typu `__wchar_t` seskupených ve znakových polích.

Charakteristika odkazového datového typu System::Object

Základy datového typu `System::Object` spočívají na řízené třídě `Object`, která je uložena ve jmenném prostoru `System`. Třída `System::Object` je primární bázovou třídou všech tříd a datových typů obecně, které mohou vývojáři v řízeném prostředí platformy .NET Framework 1.1 používat. Třída `System::Object` tedy vystupuje jako přímý, i když vzdálený, předchůdce všech dostupných datových typů, a to jak vestavěných tak uživatelsky definovaných. Poněvadž mezi jakýmkoliv datovým typem a třídou `System::Object` existuje zřetelně definovaný vztah dědičnosti, můžeme do odkazových proměnných typu `System::Object __gc*` ukládat hodnoty a reference libovolných datových typů. Tato úzká interakce mezi datovými typy a třídou `System::Object` je velice prospěšná, neboť umožňuje uskutečňovat mnoho přínosných programových operací, z kterých významnou část tvoří operace konverzního charakteru, přetěžování a parametrické substituce.

Při práci s třídou `System::Object` byste neměli zapomenout na následující důležitá fakta:

1. Třída `System::Object` reprezentuje odkazový datový typ, což znamená, že výsledkem instanciace této třídy bude předem definovaná instance, nebo jinak řečeno řízený objekt. Při požadavku na instanciaci třídy `System::Object` bude společným běhovým prostředím CLR v nulté generaci řízené hromady vyčleněný dostatečně veliký paměťový prostor pro uložení nově vytvořené instance. Odkaz na zkonstruovanou instanci třídy `System::Object` bude uložen do odkazové proměnné typu `System::Object __gc*`.

2. Na každou instanci třídy `System::Object` musí být nasměrována alespoň jedna řízená reference, která bude uložena v odkazové proměnné typu `System::Object __gc*`. Přestože je možné deklarovat odkazovou proměnnou vzpomenutého typu, aniž by došlo k instanciaci třídy `System::Object`, opačně tento proces nefunguje. Jednoduše není možné vytvořit takou instanci třídy `System::Object`, na kterou by nebyla nasměrována žádná reference jisté odkazové proměnné typu `System::Object __gc*`.

3. Odkazová proměnná typu `System::Object __gc*` může být inicializována libovolnou hodnotou hodnotového i odkazového datového typu. V prvním případě, když půjde o přiřazovací vztah **odkazová proměnná typu System::Object __gc* = hodnota hodnotového datového typu**, je nutné na hodnotu hodnotového typu aplikovat klíčové slovo `__box` (jinak by kompilátor jazyka C++ s Managed Extensions vygeneroval chybové hlášení). Klíčové slovo `__box` aktivizuje mechanizmus sjednocení typů, jenž zabezpečí založení řízeného objektu s exaktní kopií hodnoty hodnotového datového typu. Odkaz na sestrojený řízený objekt je poté uložen do odkazové proměnné typu `System::Object __gc*`. Ve druhém případě, jde-li o přiřazovací vztah **odkazová proměnná typu System::Object __gc* = reference odkazového typu**, není mechanizmus sjednocení typů inicializován. Za těchto podmínek dochází ke kopírování řízené reference směřující na instanci zdrojového odkazového datového typu, do cílové odkazové proměnné typu `System::Object __gc*`. Po ukončení kopírovacího procesu budou obě reference ukazovat na jeden a tentýž objekt odkazového datového typu.

Deklarace odkazových proměnných typu System::Object __gc* a instanciace třídy System::Object

Deklarace odkazové proměnné typu `System::Object __gc*` se může uskutečnit podle několika scénářů, které jsou závislé na skutečnosti, zda chcete odkazovou proměnnou jenom deklarovat, anebo také inicializovat platnou referencí na instanci třídy `System::Object`. Pokud si budete přát provést pouhou deklaraci odkazové proměnné typu `System::Object __gc*` bez inicializace, můžete zapsat níže znázorněný deklarační příkaz.

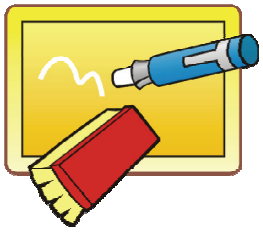
```
System::Object __gc * obj_a;
```

Kdybyste v editoru zdrojového kódu Visual C++ .NET 2003 umístili za tento deklarační příkaz lokální bod přerušení a po zastavení běhu aplikace byste se podívali do dialogového okna **Locals**, přišli byste na to, že deklarovaná odkazová proměnná `obj_a` obsahuje nedefinovanou hodnotu charakterizovanou jako `<undefined value>`. Tato informace by vás možná sváděla k úsudku, že kompilátor jazyka C++ s Managed Extensions neuskutečňuje implicitní inicializaci řízených referencí odkazových proměnných typu `System::Object __gc*`. To ovšem není pravda, protože kompilátor realizuje generování programových instrukcí, které zabezpečují inicializaci řízených ukazatelů, respektive referencí na nulovou hodnotu (popsané chování kompilátoru vychází ze standardu ISO pro jazyk C++). Pomocí následujícího fragmentu kódu můžeme vypátrat, zda byla řízená reference inicializována nulovou hodnotou či nikoliv.

```
System::Object __gc * obj_a;
if (obj_a == NULL)
   MessageBox::Show(S"obj_a == NULL");
else
   MessageBox::Show(S"obj_a != NULL");
```

Jak jistě tušíte, v okně se zprávou se objeví informační text, jenž bude říkat, že odkazová proměnná `obj_a` uchovává referenci inicializovanou nulovou hodnotou.

POZNÁMKA



Prokazatelný důkaz o požadavku na implicitní nulovou inicializaci řízených referencí můžeme nalézt při pohledu na kód jazyka Microsoft Intermediate Language (MSIL). Budeme-li předpokládat, že příkaz pro deklaraci odkazové proměnné typu `System::Object __gc*` se nachází v těle zpracovatele události `Click` instance ovládacího prvku `Button`, sestavený MSIL kód bude mít následující podobu:

```
.method private instance void  button1_Click(object
sender,
class [mscorlib]System.EventArgs e) cil managed
{
  // Code size       1 (0x1)
  .maxstack  0
  .locals init ([0] object obj_a)
  IL_0000:  ret
} // end of method Form1::button1_Click
```

V tomto výpisu si všimněte klíčové slovo `init`, které determinuje, že Just-In-Time (JIT) kompilátor musí před zpracováním programových instrukcí metody uskutečnit inicializaci všech přítomných lokálních proměnných. Pojem „uskutečnění inicializace“ je chápán z pohledu různých datových typů různě. U proměnných hodnotových datových typů znamená inicializace invokaci implicitních konstruktorů, zatímco u proměnných odkazových datových typů se pod tímto pojmem rozumí explicitní inicializace příslušných proměnných nulovými referencemi.

Deklaraci odkazové proměnné typu `System::Object __gc*` můžeme spojit s vytvořením instance třídy `System::Object` pomocí operátoru `__gc new`:

```
System::Object __gc * obj_a = __gc new System::Object();
```

V kompetenci operátoru `__gc new` je sestrojení nové instance třídy `System::Object`. Založená instance bude uložena na řízenou hromadu a běhové prostředí CLR uloží referenci na tuhle instanci do odkazové proměnné, která stojí na levé straně od přiřazovacího operátoru. Ve chvíli, kdy instance třídy `System::Object` spatří světlo světa, můžeme aktivovat některé z veřejně přístupných metod, jež instance zdědila od své mateřské třídy.

Charakteristika metod Equals, GetHashCode, GetType a ToString instance třídy System::Object

Každá instance třídy `System::Object` disponuje následujícími veřejně přístupnými instančními metodami:

- `Equals`,
- `GetHashCode`,
- `GetType`,
- `ToString`.

Abychom byli zcela přesní, musíme vzpomenout také veřejnou, ovšem statickou metodu `ReferenceEquals`, která se z funkčního hlediska ponáší na instanční metodu `Equals`. V dalším textu si vyjmenované metody představíme detailněji.

Metoda Equals

Veřejně přístupná metoda `Equals` testuje objektovou rovnost dvou instancí třídy `System::Object`, respektive instancí tříd, které jsou od této třídy přímo odvozeny. Test objektové rovnosti představuje test objektových referencí, což znamená, že metoda zjišťuje, zda jsou reference dvou odkazových proměnných nasměrované na tentýž objekt. Je-li tomu tak, metoda navrací hodnotu `true`, v opačném případě tvoří návratovou hodnotu metody hodnota `false`. Další ukázka zdrojového kódu demonstruje, jak vám může instanční metoda `Equals` pomoci při realizaci testu shody objektových referencí.

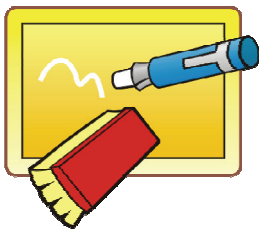
```
System::Object __gc * obj_a = __gc new System::Object();
System::Object __gc * obj_b = __gc new System::Object();

System::Boolean bShodaReferenci = obj_a->Equals(obj_b);
if (bShodaReferenci)
   MessageBox::Show(S"Objektové reference jsou shodné.", S"Zpráva",
   MessageBoxButtons::OK, MessageBoxIcon::Information);
else
   MessageBox::Show(S"Objektové reference nejsou shodné.", S"Zpráva",
   MessageBoxButtons::OK, MessageBoxIcon::Information);
```

Ze zobrazivšího dialogu se dozvíme, že objektové reference nejsou shodné. Tato skutečnost je logická, protože odkazové proměnné `obj_a` a `obj_b` obsahují reference na odlišné instance třídy `System::Object`. Další fragment programového kódu se zaměřuje na porovnávání referencí při použití přiřazovacího příkazu, v rámci kterého dochází ke kopírování reference zdrojové odkazové proměnné typu `System::String __gc*` do cílové odkazové proměnné typu `System::Object __gc*`.

```
System::Object __gc * obj_a = __gc new System::Object();
System::String __gc * str1 = System::Environment::get_UserName();

obj_a = str1;

System::Boolean bShodaReferenci = obj_a->Equals(str1);

if (bShodaReferenci)
      MessageBox::Show(S"Objektové reference jsou shodné.", S"Zpráva",
      MessageBoxButtons::OK, MessageBoxIcon::Information);
else
      MessageBox::Show(S"Objektové reference nejsou shodné.", S"Zpráva",
      MessageBoxButtons::OK, MessageBoxIcon::Information);
```

Odkazová proměnná `str1` typu `System::String __gc*` pracuje s referencí, která směruje na instanci třídy `System::String` (tato instance obsahuje textový řetězec identifikující uživatelské jméno aktuálně přihlášeného uživatele). Všimněte si, že když použijeme přiřazovací příkaz `obj_a = str1;`, kompilátor jazyka C++ s Managed Extensions provede bez jakýchkoliv problémů kopírování reference z proměnné **str1** do proměnné `obj_a`. To znamená, že obě odkazové proměnné od této chvíle obsahují řízenou referenci na identickou instanci. Řečeno jinými slovy, objektové reference obou proměnných jsou totožné.

POZNÁMKA



Bázová knihovna tříd vývojově-exekuční platformy .NET Framework 1.1 definuje dvě varianty metody `Equals`, které se liší jednak svou povahou a taktéž signaturou, tedy počtem deklarovaných formálních parametrů. První verzi metody `Equals` lze použít pouze s instancemi třídy `System::Object`, nebo s instancemi tříd, které jsou od třídy `System::Object` odvozeny. (Jelikož v zásadě všechny třídy jsou implicitně odvozeny od primární bázové třídy `System::Object`, můžeme prohlásit, že instanční metodu `Equals` je možné použít ve spojení s instancí libovolné vestavěné nebo uživatelsky definované třídy.) Kromě toho však existuje i statická verze metody `Equals`, kterou lze zavolat, aniž bychom byli nuceni vytvářet novou instanci třídy `System::Object`. Statická metoda `Equals` disponuje dvěma parametry, které přijímají reference typu `System::Object __gc*`, přičemž test shodnosti referencí je realizovaný na dodaných argumentech. Aktivaci instanční metody `Equals` smíme snadno nahradit voláním statické verze této metody. Níže uvedené řádky programového kódu jsou proto funkčně ekvivalentní.

```
// Tento řádek používá instanční formu metody Equals...
System::Boolean bShodaReferenci_1 = obj_a->Equals(str1);
// ...zatímco na následujícím řádku se ke slovu dostává
// statická verze stejnojmenné metody.
System::Boolean bShodaReferenci_2 =
System::Object::Equals(obj_a, str1);
```

Instanční varianta metody `Equals` je definována jako virtuální, což znamená, že jestli budete mít chuť, můžete v odvozené třídě tuto metodu překrýt a implementovat tak vlastní programovou funkcionalitu.

Metoda GetHashCode

Virtuální veřejně přístupná instanční metoda `GetHashCode` dovoluje získat přístup k symbolovému kódu instance třídy `System::Object`. Symbolový kód nebo také heš kód nachází své uplatnění především při začleňování instancí do asociativních tabulek. Symbolový kód platný pro instanci třídy `System::Object` vrací metoda `GetHashCode` v podobě své návratové hodnoty typu `System::Int32`. Informaci o symbolovém kódu vygenerované instance třídy `System::Object` můžete získat pomocí tohoto fragmentu programového kódu:

```
System::Object __gc * obj_a = __gc new System::Object();
System::Int32 SymbolovyKod = obj_a->GetHashCode();
MessageBox::Show(System::String::Concat(S"Symbolový kód instance je ",
   SymbolovyKod.ToString(), S"."), S"Symbolový kód instance",
   MessageBoxButtons::OK, MessageBoxIcon::Information);
```

Metoda GetType

Zavoláte-li veřejně přístupnou metodu `GetType` instance třídy `System::Object`, získáte návratovou hodnotu v podobě instance třídy `Type` z jmenného prostoru `System`. Vrácená

instance třídy `System::Type` reprezentuje objekt, který vám prostřednictvím reflexe dovede podat relevantní informace nejenom o analyzované instanci třídy `System::Object`, ale v podstatě o všech datových typech, jež jsou deklarovány ve zkoumaném sestavení. Instance třídy `System::Type` je při dolování informací o datových typech, tzv. metadatech, velice užitečná a v mnoha případech takřka neocenitelná. V následujícím výňatku kódu naznačujeme cestu, jak vytěžíte informace o názvu a umístění sestavení, ve kterém je definována třída `System::Object`.

```
System::Object __gc * obj_a = __gc new System::Object();
System::Type __gc * TypInstance = obj_a->GetType();

System::Reflection::Assembly __gc * obj_Sestaveni =
   TypInstance->get_Assembly();

System::String __gc * strInfoOSestaveni = System::String::Concat(
   S"Informace o sestavení: ", obj_Assembly->get_FullName(),
   System::Environment::NewLine,
   S"Umístění sestavení: ", obj_Assembly->get_Location());

MessageBox::Show(strInfoOSestaveni, S"Informace o sestavení",
   MessageBoxButtons::OK, MessageBoxIcon::Information);
```

Metoda ToString

Poslední veřejně přístupnou instanční metodou třídy `System::Object`, se kterou se seznámíme, je metoda `ToString`. Ta vrací textovou identifikaci instance datového typu, která je uživatelsky přívětivá a navíc bere v potaz také konfigurační nastavení lokálního prostředí operačního systému. Metoda `ToString` je virtuální, a proto ji mohou programátoři překrýt a zabezpečit tak požadovaný formát výstupních textových dat. Ve spojení s instancí třídy `System::Object` může mít použití metody `ToString` třeba tuto podobu:

```
System::Object __gc * obj_a = new System::Object();
MessageBox::Show(obj_a->ToString(), S"Textová informace o objektu",
   MessageBoxButtons::OK, MessageBoxIcon::Information);
```

Charakteristika mechanizmu sjednocení typů

Mechanizmus sjednocení typů umožňuje programátorům provádět účinnou konverzi instancí hodnotových datových typů do podoby objektů odkazových typů. Srdcem mechanizmu sjednocení typů je technika zabalení dat, která iniciuje přetypování instance jistého hodnotového datového typu na instanci odkazového datového typu. Zjednodušeně bychom mohli říci, že vstupní surovinou pro mechanizmus sjednocení typů je hodnota hodnotového typu, která je následně umístěna do nově vytvořené instance odkazového typu. Programátoři pracující s jazykem C++ s Managed Extensions musejí mechanizmus sjednocení typů aktivovat pokaždé, když je zapotřebí uskutečnit vzájemnou interakci mezi hodnotovými a odkazovými datovými typy. Příkladem takovéto interakce může být přiřazení hodnoty instance hodnotového typu do proměnné odkazového typu `System::Object __gc*`, či potřeba uložení hodnoty instance hodnotového typu do parametru generického odkazového typu funkce nebo metody.

Hodnotové a odkazové datové typy formují základní stavební strukturu typů platformy .NET Framework. Ačkoliv mezi instancemi obou kategorií typů dochází k vzájemným vztahům, obě skupiny typů se liší v několika podstatných aspektech, o nichž pojednáme dále.

Instance hodnotových datových typů představují plně vybavené kontejnery, jež jsou připraveny na explicitní uchování libovolné konkrétní hodnoty z definičního oboru deklarovaného datového typu. Pokud programátor uskuteční deklaraci proměnné hodnotového typu, jakým je například `System::Byte`, `System::Int16` nebo `System::Double`, na zásobníku bude alokován dostatečný paměťový prostor pro uložení hodnoty právě deklarované proměnné. Hodnotová proměnná působí jako instance hodnotového typu. Jestli do proměnné uložíme v rámci přiřazovacího příkazu smysluplnou hodnotu, tato je okamžitě uložena do předem připravené paměťové oblasti. Důležité je uvědomit si, že instance hodnotového datového typu, tedy hodnotová proměnná, přímo obsahuje přiřazená data. Všechny instance hodnotových datových typů jsou v rámci společného běhového prostředí CLR ukládány na zásobník, kromě jediné výjimky, k níž dojde, když je instance hodnotového typu součástí instance odkazového typu.

Dalším aspektem použití instancí hodnotových datových typů, je jejich chování při kopírování dat z jedné instance do druhé. Povězme, že máme k dispozici dvě instance hodnotového datového typu. Co se stane, když použijeme přiřazovací příkaz `HodnotováInstance1 = HodnotováInstance2;`? Nuže, dojde ke kopírování hodnoty druhé instance do instance první. Přitom jsou zkopírována všechna nezbytná data, což znamená, že po zpracování uvedeného příkazu budou obě instance naplněny stejnými hodnotami. Samozřejmě, pokud se budou přítomné instance různit v deklarovaných hodnotových datových typech, je možné, že kompilátor jazyka C++ s Managed Extensions bude muset vykonat určitý počet konverzních operací. Kompilátor si však poradí jedině s implicitně realizovanými konverzemi. Nebude-li schopen sám konverzi uskutečnit, zobrazí chybové hlášení. V tomto momentě dochází k nesouladu datových typů proměnných a do hry se mohou zapojit explicitní konverzní techniky.

Dostane-li se hodnotová proměnná mimo svůj obor platnosti, je automaticky odstraněna ze zásobníku a podrobena destrukci. Důležité je poukázat na skutečnost, že proces likvidace instance hodnotového typu je zcela deterministický, což znamená, že jej lze přesně časově vymezit. O alokaci a dealokaci zdrojů asociovaných s instancí hodnotového typu se stará běhové prostředí CLR ve spolupráci se zásobníkem.

Z uvedeného plyne, že pro instance hodnotových typů platí tyto zásady:

1. Instance hodnotového typu je vytvořena okamžitě poté, co je deklarována proměnná tohoto typu.

2. Instance hodnotového typu sídlí v rezervované paměťové oblasti řízeného procesu, které se říká zásobník.

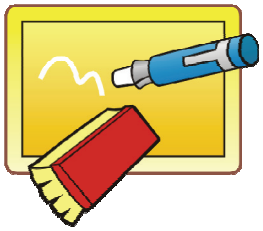
3. Instance hodnotového typu obsahuje vždy konkrétní data (a ne odkazy na tato data).

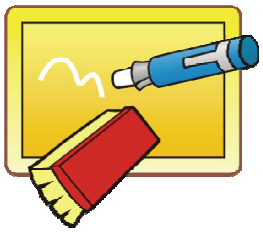
4. Přiřazovací příkaz `HodnotováInstance1 = HodnotováInstance2;`, v němž jsou typy obou instancí hodnotové datové typy, spouští kopírovací proces, jenž provádí duplikaci hodnoty zdrojové instance a tuhle následně ukládá do cílové instance. Je možné, že tento proces bude vyžadovat aktivaci konverzních operací.

5. Životní cyklus instance hodnotového typu je úzce spojen s výskytem hodnotové proměnné v syntaktických blocích programu. Jakmile hodnotová proměnná překročí svůj obor působnosti, její životní cyklus se končí a proměnná, respektive instance hodnotového typu umírá. Likvidace instance hodnotového typu pracuje podle přesně stanoveného algoritmu a je tudíž predikovatelná.

U instancí odkazových datových typů je situace o poznání komplikovanější. Mnohé, dokonce i zkušené programátory mate skutečnost, že v tomto případě nemůžeme

přijmout implikaci „proměnná odkazového typu == instance odkazového typu". Ve skutečnosti jsou odkazová proměnná a instance odkazového typu dvě rozdílné entity. Zatímco pro deklaraci proměnné odkazového typu platí podobná pravidla jako pro vytvoření proměnné hodnotového typu, instanci odkazového typu můžeme explicitně založit pouze prostřednictvím operátoru `new`, respektive `__gc new`.

POZNÁMKA	
	Sestrojení instance odkazového typu se však může řídit i podle implicitního pracovního modelu, když práci spojenou se založením instance na sebe převezme určitá funkce nebo metoda (buď instanční nebo statická). Pokud klientsky programový kód aktivuje takovouto funkci, funkce vytvoří kýženou instanci a vrátí `__gc` ukazatel na tuto instanci. To znamená, že programátor není nucen explicitně používat operátor `__gc new` pro zkonstruování nové instance odkazového typu, přestože vyvolaná metoda tento operátor interně zcela jistě aplikuje.

POZNÁMKA	
	Jazyk C++ s Managed Extensions nevyžaduje explicitní použití operátoru `__gc new`. Ve skutečnosti mohou programátoři při vytváření nové instance odkazového datového typu použít rovněž obecný operátor `new`. Naštěstí, kompilátor je natolik inteligentní, že dovede sám zjistit, či specifikovaný datový typ splňuje požadavky kladené na odkazové datové typy bázové knihovny tříd platformy .NET Framework 1.1 a vybere korektní řízenou verzi operátoru `new`, jejíž syntaktická podoba je `__gc new`.

Proměnná odkazového datového typu je vytvořena na zásobníku, kde je pro ni alokován kapacitně vyhovující paměťový prostor. Proměnná dále „čeká", dokud operátor `__gc new` nevykoná všechny nezbytné operace, které si vyžaduje založení nové instance odkazového typu. Poté je do odkazové proměnné uložen `__gc` ukazatel na zrozenou instanci.

Kdybychom se na práci operátoru `__gc new` podívali blíže, došli bychom k poznání, že uvedený operátor provádí tyto činnosti:

1. Pro nově vytvářenou instanci odkazového datového typu operátor `__gc new` alokuje dostatečný prostor v nulté generaci řízené hromady. Řízená hromada je soustavou tří objektových generací s pořadovými čísly 0, 1 a 2, do nichž jsou ukládány instance odkazových typů v závislosti na stadiu jejich životních cyklů. Všechny nové instance jsou nejprve umísťovány do generace č. 0 řízené hromady. Možná víte, že řízenou hromadu pečlivě kontroluje softwarová služba společného běhového prostředí CLR s názvem automatická správa paměti (angl. Garbage Collection, GC). Služba GC sestavuje pro každou instanci odkazového typu strom referencí, podle kterého je v kterémkoliv okamžiku možné jednoznačně určit, zda je jistá instance dosažitelná z programového kódu řízené aplikace anebo ne. Pokud je instance dosažitelná, je naživu a služba GC ji nemůže z operační paměti odstranit. Ve chvíli, kdy se všechny odkazové proměnné (typu `__gc*`), jež jsou namířeny na určitou instanci odkazového typu, dostanou mimo svůj obor působnosti, je dotyčná instance označena jako nedosažitelná. Takováto instance

už není samozřejmě potřebná, a proto se stává soustem pro automatickou správu paměti, která uskuteční její pozvolnou, tedy ne deterministickou dealokaci z řízené hromady.

2. Operátor `__gc new` volá instanční konstruktor, jenž provede inicializaci datových položek pro nově založenou instanci.

3. Operátor `__gc new` vrací řízený ukazatel na zrozenou instanci, jenž je uložen do odkazové proměnné spočívající na zásobníku.

Zapíšeme-li příkaz `OdkazováProměnná1 = OdkazováProměnná2;`, přičemž budeme předpokládat, že obě odkazové proměnné byly deklarovány pomocí stejného odkazového datového typu, pak můžeme říci, že operátor přiřazení iniciuje kopírování reference ze zdrojové proměnné (`OdkazováProměnná2`) do cílové proměnné (`OdkazováProměnná1`). Je důležité, abyste si uvědomili, že po zpracování přiřazovacího příkazu budou obě odkazové proměnné obsahovat referenci identifikující totožnou instanci odkazového datového typu.

Při práci s odkazovými proměnnými a instancemi odkazových typů musíme mít na paměti následující zásady:

1. Proměnná obsahující referenci na instanci odkazového typu a samotná instance tohoto typu jsou dvě různé entity.

2. Odkazová proměnná je uložena na zásobníku, zatímco instance odkazového typu je lokalizována na řízené hromadě běhového prostředí CLR.

3. Při přiřazení hodnoty jedné odkazové proměnné do jiné odkazové proměnné dochází ke kopírování referencí.

4. Instance odkazového typu není zlikvidována neprodleně poté, co se odkazová proměnná dostane mimo svůj obor platnosti. Životní cyklus instance odkazového typu sleduje automatický správce paměti, který zahájí proces destrukce vybrané instance až v okamžiku, kdy se tato stane nedosažitelnou z programového kódu.

Všechny popsané poznatky nyní využijeme při důkladném studiu pracovního postupu mechanizmu sjednocení typů.

Algoritmus činnosti mechanizmu sjednocení typů

Mechanizmus sjednocení typů je nutno aktivovat pokaždé, když budeme chtít s instancí hodnotového typu pracovat jako s objektem. S požadavkem na spuštění mechanizmu sjednocení typů se setkáme například v níže uvedeném fragmentu programového kódu, který provádí přiřazení hodnoty pseudonáhodného celého čísla typu `System::Int32` do odkazové proměnné typu `System::Object __gc*`.

```
System::Random __gc * NahodneCislo = __gc new System::Random();
System::Object __gc * obj_NahodneCislo = NahodneCislo->Next(1, 101);
MessageBox::Show(System::String::Concat(S"Bylo vygenerováno číslo ",
   obj_NahodneCislo->ToString(), S"."), S"Náhodné číslo",
   MessageBoxButtons::OK, MessageBoxIcon::Information);
```

Jestliže bychom vydali příkaz na přeložení tohoto kódu, za několik málo okamžiků by nás kompilátor jazyka C++ s Managed Extensions upozornil na druhý řádek kódu se zprávou *„error C2440: initializing: cannot convert from int to System::Object __gc*"*. Kompilátor oznamuje, že není schopen konvertovat hodnotu typu `System::Int32` (kterou vrací metoda `Next` instance třídy `System::Random`) na objektovou referenci, jíž by bylo možné

uložit do odkazové proměnné typu `System::Object __gc*`. Inference, kterou nám kompilátor předvádí, je celkem logická: Protože odkazová proměnná může obsahovat pouze referenci na instanci jistého odkazového datového typu, není jednoduše možné do ní uložit hodnotu hodnotového datového typu. Možná byste si mohli myslet, že tento typ konverzní operace nelze vůbec provést, no to by nebyla pravda. Cesta k úspěchu není přitom vůbec klikatá – stačí, když si na pomoc zavoláme mechanizmus sjednocení typů.

Mechanizmus sjednocení typů aktivizujeme pomocí příkazu `__box`:

```
System::Object __gc * obj_NahodneCislo = __box(NahodneCislo->Next(1, 101));
```

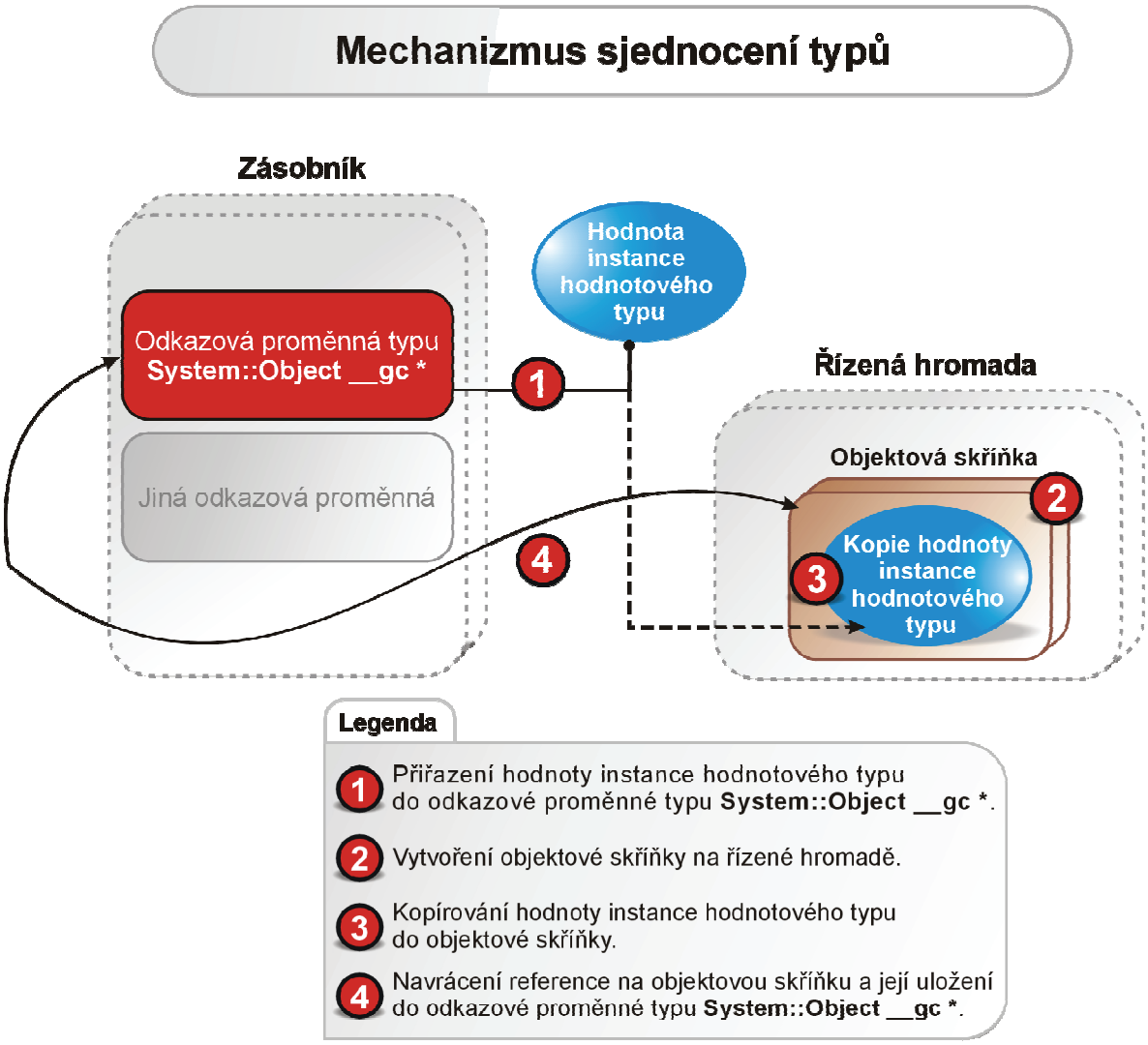
A co vlastně příkaz `__box` dělá? Přibližně toto:

1. V nulté generaci řízené hromady alokuje dostatečně velký prostor, do něhož vzápětí uloží objektovou skříňku. Pod termínem objektová skříňka rozumíme kontejner, jenž je schopen přijmout kopii dat zdrojové instance hodnotového datového typu.

2. Nastartuje kopírovací proces, který realizuje kopírování dat instance hodnotového typu do objektové skříňky nacházející se na řízené hromadě. Výsledkem procesu kopírování je uložení přesného bitového obrazu hodnoty instance hodnotového typu do objektové skříňky. Nyní tedy objektová skříňka obsahuje tutéž hodnotu jako instance hodnotového typu, na kterou byl příkaz `__box` aplikován.

3. Vrací objektovou referenci směřující na objektovou skříňku. Tato reference je uložena do odkazové proměnné typu `System::Object __gc*`.

Nuže, vážení přátelé, takhle probíhá mechanizmus sjednocení typů. Jeho finálním produktem je konstrukce objektové skříňky s kopií hodnoty zdrojové instance hodnotového datového typu. Mezi hodnotou původní instance hodnotového typu a její binární reprezentací působící uvnitř objektové skříňky neexistuje žádný vzájemný vztah. To tedy znamená, že pokud změníte hodnotu instance hodnotového typu, tato změna se automaticky nepromítne do obsahu objektové skříňky. Grafickou ilustraci pracovního postupu mechanizmu sjednocení typů ilustruje obr. 1.2.



Obr. 1.2: Schematické znázornění pracovního algoritmu mechanizmu sjednocení typů

Programovací jazyk C++ s Managed Extensions neuskutečňuje implicitní spuštění mechanizmu sjednocení typů. Je to způsobeno tím, že samotný průběh mechanizmu sjednocení typů vyžaduje zpracování dodatečných programových instrukcí, které mohou v jistých případech zapříčinit nemalou výkonnostní penalizaci. Není pochyb o tom, že nejkritičtější etapou činnosti mechanizmu sjednocení typů je z hlediska výkonu právě alokace objektové skříňky na řízené hromadě a kopírování dat do této skříňky. Řízené C++ proto vyžaduje od programátora výslovný příkaz, a až poté aktivuje mechanizmus sjednocení typů. V tomto směru se C++ s Managed Extensions liší od jiných .NET-kompatibilních programovacích jazyků (jako je Visual Basic .NET a C#), nakolik tyto jazyky provádějí sjednocení typů implicitně, tedy kdykoliv je to zapotřebí (programátor přitom ani nemusí vědět, že na daném místě dochází ke spuštění mechanizmu, jenž sjednocuje zúčastněné datové typy).

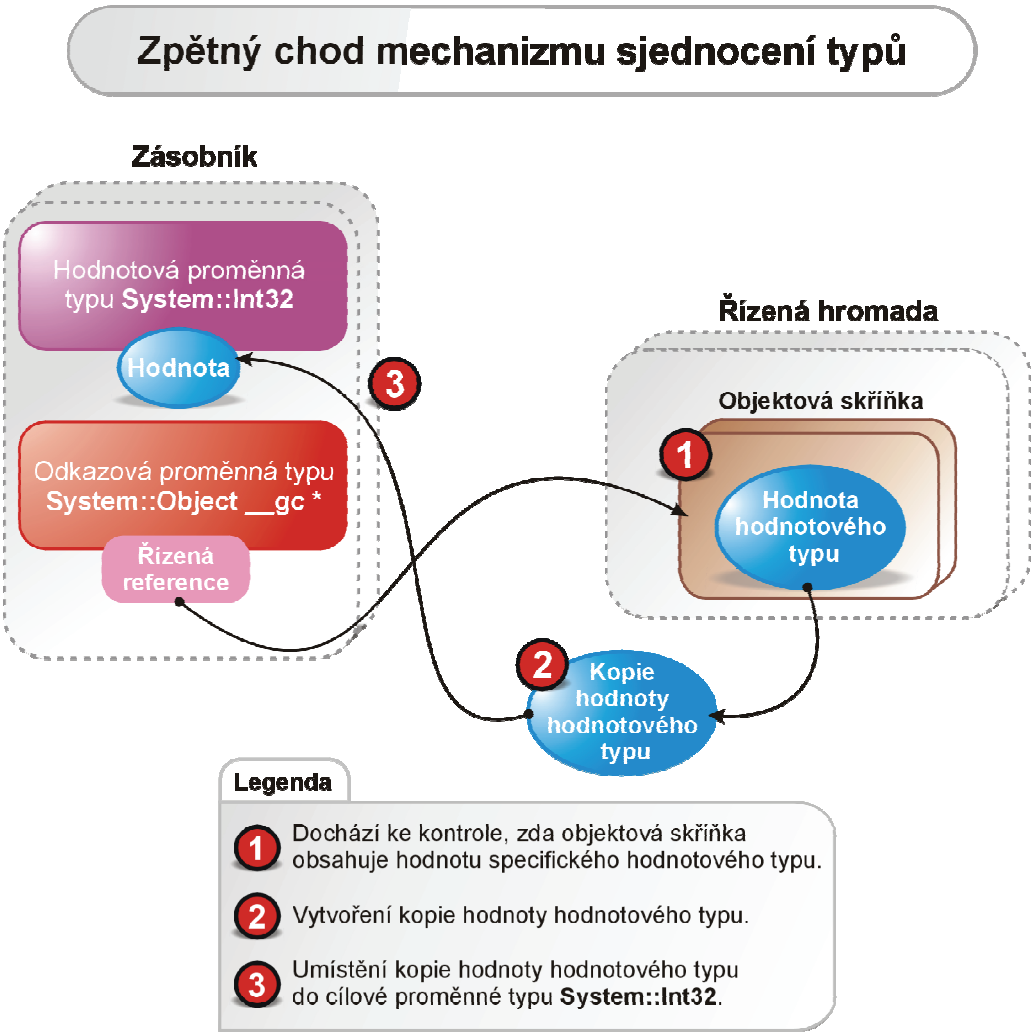
Charakteristika zpětného chodu mechanizmu sjednocení typů

V jistých případech je nevyhnutné získat hodnotu z objektové skříňky zpět a použít ji při uskutečňování dalších programových operací. Tato posloupnost akcí se označuje jako zpětný mechanizmus sjednocení typů (angl. unboxing). Bohužel, ve výbavě jazyka C++ s Managed Extensions není zastoupen žádný „opozitní" příkaz k příkazu `__box`, a proto si

musíme pomoci vlastními silami. Následující výpis kódu předvádí použití zpětného chodu mechanizmu sjednocení typů pomocí operátoru `dynamic_cast<>`.

```
System::Random __gc * NahodneCislo = __gc new System::Random();
System::Object __gc * obj_NahodneCislo = __box(NahodneCislo->Next(1, 101));
System::Int32 Cislo =
   *dynamic_cast<__box System::Int32 __gc*>(obj_NahodneCislo);
MessageBox::Show(System::String::Concat(S"Bylo vygenerováno číslo ",
   Cislo.ToString(), S"."), S"Náhodné číslo",
   MessageBoxButtons::OK, MessageBoxIcon::Information);
```

Zpětný běh mechanizmu sjednocení typů demonstruje třetí řádek zobrazeného zdrojového kódu. V něm je aplikovaný již zmíněný operátor `dynamic_cast<>`, jehož prostřednictvím zjišťujeme, zda odkazová proměnná `obj_NahodneCislo` typu `System::Object __gc*` uchovává referenci na objektovou skříňku s hodnotou typu `System::Int32`. Pokud je takto stanovená podmínka splněna, můžeme získat `__gc` ukazatel směřující na hodnotu typu `System::Int32`, a tento poté dereferencovat, čímž se dostaneme ke skutečné hodnotě uložené v objektové skříňce. V další etapě hodnotu uloženou v objektové skříňce zkopírujeme do připravené proměnné `Cislo` typu `System::Int32`. Grafickou podobu zpětného chodu mechanizmu sjednocení typů přináší obr. 1.3.



Obr. 1.3: Grafická ilustrace zpětného chodu mechanizmu sjednocení typů

Řízené třídy (__gc třídy)

Řízené třídy představují programové šablony, které exaktně definují syntaktickou stavbu a charakter chování svých instancí, jimiž jsou programové objekty řízené společným běhovým prostředím CLR platformy .NET Framework 1.1. Řízenou třídu ve zdrojovém kódu poznáte velice snadno, protože každá takováto třída musí mít ve své hlavičce uvedeno klíčové slovo `__gc` (mimochodem, „gc" je zkratka pro „garbage-collected", což znamená, že instance řízené třídy jsou pod kontrolou automatické správy paměti). Rozhodneme-li se vstoupit do světa řízených tříd, měli bychom pamatovat na několik důležitých postulátů:

1. Jakákoliv řízená třída je uživatelsky definovaným odkazovým datovým typem, přičemž instance `__gc` třídy jsou řízené běhovým prostředím CLR. Abychom mohli jakoukoliv třídu prohlásit za řízenou, je nutné, aby byla takováto třída definována pomocí modifikátoru `__gc`.

2. Vedle řízených tříd dovoluje jazyk C++ s Managed Extensions programátorům definovat také hodnotové (`__value`) třídy. Jelikož `__value` třídy jsou na rozdíl od `__gc` tříd hodnotovými datovými typy, můžeme mezi vzpomenutými kategoriemi tříd vyhledat více rozdílů:

   - Zaprvé, `__value` třídy jsou odvozeny od bázové třídy `System::ValueType`, zatímco mateřskou třídou pro všechny `__gc` třídy je třída `System::Object`.

   - Zadruhé, instance hodnotových tříd jsou primárně určeny pro práci s kapacitně nenáročnými daty a jejich životní cyklus je ve srovnání s instancemi řízených tříd o poznání kratší.

   - Zatřetí, instance hodnotových tříd jsou ukládány zpravidla na zásobník a ne na řízenou hromadu, jak je tomu u instancí řízených tříd. Vzhledem k tomu, že zásobník je efektivně spravovanou datovou strukturou založenou na principu LIFO, je alokace a dealokace instancí hodnotových tříd méně komplikovaná a hlavně deterministická. To znamená, že víme relativně přesně vymezit okamžik, kdy bude instance `__value` třídy zrozena a kdy bude její životní cyklus ukončen.

     Třebaže si instance hodnotových tříd se zásobníkem dobře rozumějí, v praxi se mohou vyskytnout rovněž situace, kdy budou tyto instance alokovány na řízené hromadě a nikoliv na zásobníku. Příkladem je mechanizmus sjednocení typů, při kterém se na řízené hromadě konstruuje objektová reprezentace instance hodnotového typu. Instance `__value` třídy bude na řízenou hromadu umístěna také tehdy, pokud je definice hodnotové třídy součástí definice řízené třídy, respektive tehdy, dochází-li v těle řízené třídy k instanciaci hodnotové třídy. Konečně, instance `__value` třídy se na řízené hromadě octne také tehdy, když je tato instance uložena v řízeném poli. Naproti tomu, instance řízené třídy jsou vždy zakládány na řízené hromadě, která je ovládána automatickým správcem paměti.

TIP

Instance hodnotové třídy může být na rozdíl od `__gc` třídy alokována také na standardní hromadě C++, tedy ne na řízené hromadě běhového prostředí CLR. V tomto případě však musíme při instanciaci `__value` třídy explicitně použít operátor `__nogc new`. Jelikož za těchto okolností není instance automaticky uvolněna z paměti, je jí nutno podrobit explicitní destrukci prostřednictvím operátoru `delete`. Níže je zapsán programový kód, jenž uskutečňuje definici a posléze také instanciaci hodnotové třídy `PrestupnyRok`. Parametrický konstruktor této třídy na základě dodaného argumentu zjišťuje, zda daný argument charakterizuje přestupný rok.

```
#include "stdafx.h"
#using <mscorlib.dll>
using namespace System::Windows::Forms;
public __value class PrestupnyRok
{
public:
   PrestupnyRok(System::Int32 Rok)
   {
      System::Boolean bRok =
         System::DateTime::IsLeapYear(Rok);
      if (bRok)
         MessageBox::Show(System::String::Concat(S"Rok ",
         Rok.ToString(), S" je přestupný."));
      else
         MessageBox::Show(System::String::Concat(S"Rok ",
         Rok.ToString(), S" není přestupný."));
   }
};

...

PrestupnyRok __nogc * obj_A =
   __nogc new PrestupnyRok(2004);
delete obj_A;
```

Protože jsme při vytváření objektu hodnotové třídy použili operátor `__nogc new`, zrozený objekt bude alokován na standardní hromadě. Neřízený (`__nogc*`) ukazatel na sestrojený objekt bude uložen do odkazové proměnné `obj_A` typu `PrestupnyRok __nogc*`. Konstruktor objektu přijímá argument reprezentující rok v podobě hodnoty datového typu `System::Int32`. V okamžiku, když instanci hodnotové třídy již nebudeme potřebovat, uvolňujeme její zdroje voláním operátoru `delete`.

3. Výsledným produktem instanciace řízené třídy je objekt, jenž je alokován na řízené hromadě procesu aplikace .NET. Aby mohl být vytvořený objekt dosažitelný z programového kódu, je konstruktorem instance vrácený řízený (`__gc*`) ukazatel, jenž je uložen do odkazové proměnné typu `T __gc*`, kde `T` je datový typ instance třídy, na níž je řízený ukazatel nasměrován. Řízené ukazatele jsou charakteristické následujícími vlastnostmi:

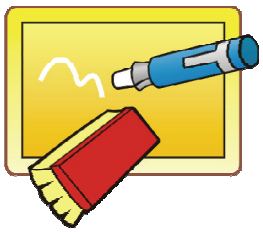
- Řízený ukazatel smí být nasměrován pouze na platnou instanci řízené třídy, která je alokovaná na řízené hromadě běhového prostředí CLR.

- Řízený ukazatel je typově silným ukazatelem, což znamená, že může mířit jedině na instanci determinovaného datového typu.

- Jazyk C++ s Managed Extensions rozlišuje dvě hlavní skupiny řízených ukazatelů:

    - ukazatele, které směrují na „celé" objekty,
    - ukazatele, které směrují na podobjekty řízených (`__gc`) tříd, anebo objekty hodnotových (`__value`) tříd.

- Automatický správce paměti sleduje všechny řízené ukazatele, jež míří na jistou instanci řízené třídy. Správce paměti vytváří na pozadí referenční strom řízených ukazatelů a příslušných instancí nacházejících se na řízené hromadě. Podle rozvoje referenčního stromu řízených ukazatelů je možné kdykoliv spolehlivě určit, zda je konkrétní instance `__gc` třídy dosažitelná z programového kódu nebo ne. Není-li na instanci nasměrován ani jeden `__gc` ukazatel, správce paměti nařídí uvolnit tuto instanci a dealokovat obsazené systémové zdroje. Jak si můžete všimnout, automatická správa objektů se liší od pracovního modelu nativního C++, neboť zbavuje programátora povinnosti zavolání operátoru `delete`, který v nativním světe likviduje dynamicky zrozenou třídní instanci. Správa životních cyklů objektů .NET se však liší také od kontroly doby životnosti objektů COM, která je řízena algoritmem automatického počítání referencí ve vztahu k dostupným nativním ukazatelům na příslušná rozhraní.

POZNÁMKA	
	V prostředí COM sehrávají v tomto směru důležitou roli virtuální funkce `AddRef` a `Release` rozhraní `IUnknown`, které implementuje každá třída COM. Funkce `AddRef` uskutečňuje inkrementaci datového členu, jenž uchovává počet referencí na objekt COM. Na druhé straně, funkce `Release` realizuje dekrementaci hodnoty vzpomenutého datového členu a v další etapě zjišťuje, zda je počet referencí na objekt COM roven nule. Jestli ano, znamená to, že na objekt nejsou navázány žádné reference, a proto může být z operační paměti odstraněn. Pokud ne, životní cyklus objektu se ještě nekončí a objekt tak i nadále zůstává naživu.

  Finalizace instancí `__gc` tříd na řízené hromadě prostřednictvím správce paměti není časově jednoznačná, čili nedovedeme na vteřinu přesně říci, kdy bude ta-která instance zlikvidována a její zdroje uvolněny.

- Odkazová proměnná typu `System::Object __gc*` může uchovávat řízený ukazatel na instanci libovolné `__gc` třídy.

- Řízené ukazatele jsou implicitně inicializované na nulové hodnoty, což znamená, že programátor nemusí tuto inicializaci provádět ve vlastní režii.

4. Založený objekt řízené třídy splňuje všechny náležitosti, které jsou na něj ze strany společné jazykové infrastruktury vývojově-exekuční platformy .NET Framework 1.1 kladeny. Objekt je tak schopen poskytnut informace o svých

soukromých datových členech prostřednictvím programových vlastností, dále může reagovat na události a vykonávat operace pomocí programových metod. Objekt je přitom plně samostatní a nezávislá softwarová entita, která dokáže zaujmout roli abstraktního substitutu jakéhokoliv předmětu skutečného (fyzického) světa.

Vytváříme první řízenou (__gc) třídu v jazyce C++ s Managed Extensions

Abychom si předvedli tvorbu řízené třídy v praxi, vytvoříme si jednu sami: půjde o `__gc` třídu s názvem `Kniha`, která bude představovat jednoduchou virtuální reprezentaci knižní publikace. Třídu obohatíme o dvě privátní datové položky, jejichž pomocí budeme uchovávat informace o názvu a počtu stran publikace. Aby mohl klientský programový kód číst a případně také upravovat hodnoty datových položek naší třídy, zařadíme do její definice příslušné vlastnosti, které budeme implementovat dvojící metod `get_` a `set_`. V těle řízené třídy `Kniha` se bude nacházet veřejně přístupný parametrický instanční konstruktor, jenž bude realizovat počáteční inicializaci soukromých datových položek instance třídy. Pro úspěšné vytvoření řízené třídy `Kniha` postupujte prosím dle následujících instrukcí:

1. Jestliže jste tak ještě neučinili, spusťte produkt Visual C++ .NET 2003 a vytvořte nový projekt standardní aplikace pro systém Windows (**Windows Forms Application (.NET)**).

2. Otevřete nabídku **Project** a klepněte na položku **Add New Item**, nebo vyvolejte klávesovou zkratku CTRL+SHIFT+A.

3. V zobrazeném dialogovém okně vyhledejte položku **C++ File (.cpp)**, která slouží pro založení nového implementačního souboru jazyka C++.

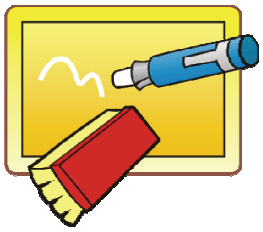
4. Do přidaného souboru vložte nevyhnutné direktivy společně s řádky zdrojového kódu pro definici jmenného prostoru a řízené třídy `Kniha`:

```
#include "stdafx.h"
#using <mscorlib.dll>
using namespace System::Windows::Forms;
// Definice jmenného prostoru, ve kterém je __gc třída uložena.
namespace GC_Tridy
{
// Definice řízené třídy.
public __gc class Kniha
{
private:
   // Deklarace soukromých datových položek __gc třídy.
   System::String __gc * _NazevKnihy;
   System::UInt16 _PocetStran;
public:
   // Definice veřejně přístupného parametrického konstruktoru.
   // Konstruktor realizuje inicializaci privátních datových položek.
   Kniha(System::String __gc * NazevKnihy,
   System::UInt16 PocetStran) :
   _NazevKnihy(NazevKnihy), _PocetStran(PocetStran){}

   // Definice metody get_ vlastnosti NazevKnihy.
   __property System::String __gc * get_NazevKnihy()
   {
```

```
        return _NazevKnihy;
    }
    // Definice metody set_ vlastnosti NazevKnihy.
    __property System::Void set_NazevKnihy
    (System::String __gc * NazevKnihy)
    {
        _NazevKnihy = NazevKnihy;
    }
    // Definice metody get_ vlastnosti PocetStran.
    __property System::UInt16 get_PocetStran()
    {
        return _PocetStran;
    }
    // Definice metody set_ vlastnosti PocetStran.
    __property System::Void set_PocetStran(System::UInt16 PocetStran)
    {
        _PocetStran = PocetStran;
    }
};
}
```

Řízená třída `Kniha` obsahuje soukromou i veřejnou sekci, které obě jsou označeny návěstími `private:` a `public:`. V privátní části jsou deklarovány dvě datové položky, z nichž jedna (`_NazevKnihy` typu `System::String __gc*`) bude sloužit pro uložení názvu knihy a druhá (`_PocetStran` typu `System::UInt16`) bude schopna zpracovat celočíselnou hodnotu udávající počet stran knihy. Ve veřejné sekci třídy se nachází parametrický instanční konstruktor, v jehož hlavičce jsou pomocí inicializačního seznamu inicializovány privátní datové položky třídy (vstupní inicializační data poskytují parametry konstruktoru). Novinkou jazyka C++ s Managed Extensions je použití vlastností. Ačkoliv se vlastnost z pohledu vnějšího kódu jeví pouze jako jednoduchá datová položka, ve skutečnosti je každá vlastnost reprezentována metodou `get_` (pokud je vlastnost určena pouze pro čtení), metodou `set_` (je-li vlastnost určena pouze pro zápis), anebo kombinací obou metod (jestliže je vlastnost určena pro čtení i zápis dat). Metody `get_` a `set_` jsou speciálními členskými funkcemi třídy, prostřednictvím kterých je vlastnost implementovaná. Aby bylo zřejmé, že metoda `get_`, respektive metoda `set_` zavádí jistou vlastnost, jazyk C++ s Managed Extensions emituje klíčové slovo `__property`, které dotyčnou vlastnost explicitně definuje.

V naší ukázce definujeme dvě vlastnosti: `NazevKnihy` a `PocetStran`. Zatímco vlastnost `NazevKnihy` je definována pomocí metod `get_NazevKnihy` a `set_NazevKnihy`, implementaci vlastnosti `PocetStran` mají na starosti metody `get_PocetStran` a `set_PocetStran`. Smysl práce metod `get_` a `set_` vlastností spočívá v získávání, respektive nastavování hodnot cílových datových položek třídy. Protože k těmto datovým položkám neexistuje přímý přístup, lze jejich hodnoty upravovat pouze prostřednictvím vlastností (nebo jinak řečeno za asistence členských metod, které tyto vlastnosti implementují). Datové položky třídy musejí být před vnějším kódem ukryty, neboť tento kompoziční model nám dovoluje aplikovat techniku skrývání dat, díky níž se objekt třídy jeví pro okolní svět jako „černí skříňka".

POZNÁMKA	
	Vlastnosti definované v naší ukázce řadíme do kategorie skalárních vlastností. O vlastnosti říkáme, že je skalární, pokud vyhovuje následujícím požadavkům: 1. Implementační metoda `get_` vlastnosti nedisponuje žádnými formálními parametry a vrací návratovou hodnotu typu `T`. 2. Implementační metoda `set_` vlastnosti je vybavena jedním formálním parametrem typu `T` a nevrací žádnou návratovou hodnotu. (Návratovou hodnotou metody `set_` je `void` nebo `System::Void`.)

5. Na formulář přidejte jednu instanci ovládacího prvku `Button`. Když na tuto instanci poklepáte, Visual C++ .NET 2003 vygeneruje syntaktickou kostru zpracovatele události `Click` instance.

6. Předtím, než zadáte programový kód pro instanciaci řízené třídy, je ještě potřebné vložit do hlavičkového souboru Form1.h direktivu `#include` s odkazem na náš implementační (.cpp) soubor. Direktiva má tuto podobu:

   ```
   #include "GC Trida.cpp"
   ```

7. Vytvoření nové instance řízené třídy `Kniha` a její použití je pak již docela snadné:

```
// Instanciace třídy Kniha z jmenného prostoru GC_Tridy.
GC_Tridy::Kniha __gc * MojeKniha =
__gc new GC_Tridy::Kniha(S"Úvod do počítačové grafiky", 333);

// Použití vytvořené instance.
System::String __gc * NovyRadek = System::Environment::NewLine;
MessageBox::Show(System::String::Concat(S"Název knihy: ",
   MojaKniha->get_NazevKnihy(),NovyRadek, S"Počet stran: ",
   MojaKniha->get_PocetStran().ToString()),
   S"Informace o knize", MessageBoxButtons::OK,
   MessageBoxIcon::Information);
```

Vzhledem k tomu, že o správu objektů `__gc` tříd se stará automatický správce paměti, není nutné, abychom ve chvíli, kdy již vytvořenou instanci třídy nebudeme potřebovat, používali pro její odstranění z paměti operátor `delete`.

Řízená (__gc) třída, instanční a statický konstruktor

Životní cyklus instance řízené třídy se začíná v okamžiku, kdy programátor uskuteční instanciaci této třídy prostřednictvím operátoru `__gc new`. Operátor `__gc new` alokuje na řízené hromadě společného běhového prostředí CLR prostor pro uložení instance `__gc` třídy, vytvoří novou instanci, iniciuje aktivaci konstruktoru (ať už implicitního, nebo uživatelsky definovaného) a nakonec navrátí řízený (`__gc*`) ukazatel, jenž reprezentuje referenci na instanci odkazového datového typu. Objektová reference je v dalším kroku uložena do příslušné odkazové proměnné typu `T __gc*` (řízený ukazatel na instanci typu `T`). Je zřejmé, že aplikace operátoru `__gc new` spouští řetězec mnoha dalších činností. Nejdřív se však podívejme na to, jakou úlohu v životním cyklu instanci řízené třídy sehrává konstruktor.

Konstruktor představuje členskou metodu třídy, která je zodpovědná za inicializaci privátních, respektive statických datových položek třídy. Podle charakteru členských datových položek, které jsou prostřednictvím konstruktoru inicializovány, můžeme hovořit o dvou variantách konstruktoru: jde o instanční konstruktor a statický konstruktor.

Zatímco hlavním úkolem instančního konstruktoru je uvést soukromé datové položky instance do výchozího stavu, statický konstruktor se zaměřuje na inicializaci statických datových položek třídy a svoji činnost uskutečňuje v okamžiku, kdy byl programový kód třídy úspěšně načten do operační paměti počítače. Statický konstruktor je běhových prostředím CLR volán právě jedenkrát, přičemž svou práci vykonává ještě před samotnou instanciací řízené třídy, tedy před pokusem o přístup ke statickým datovým položkám této třídy.

Charakteristika instančního konstruktoru řízené třídy

Budete-li chtít pro vaši `__gc` třídu definovat instanční konstruktor, můžete postupovat zcela stejně, jak jste to dělali v nativním C++. Také v jazyce C++ s Managed Extensions vystupuje instanční konstruktor jako členská metoda třídy, která se vyznačuje níže uvedenými specifiky:

1. Instanční konstruktor musí mít stejný název jako řízená třída, v jejímž těle je jeho programový kód uložen.

2. Instanční konstruktor musí být veřejně přístupným datovým členem řízené třídy. To znamená, že definice instančního konstruktoru se musí nacházet ve veřejné části třídy (tato část je vizuálně označena návěstím `public:`). Pokud by konstruktor nebyl veřejně přístupný, nemohl by se podílet na sestavování instancí `__gc` třídy. Řízená třída by tak nemohla vytvářet své vlastní instance a stala by se prakticky nepoužitelnou.

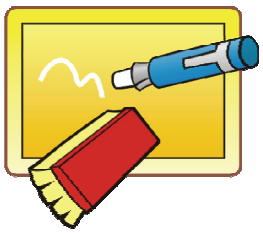
3. Instanční konstruktor nesmí disponovat žádnou specifikací své návratové hodnoty (nelze použít ani klíčové slovo `void`, ani hodnotovou strukturu `System::Void`). Jednoduše řečeno, konstruktor nepracuje s žádnou návratovou hodnotou.

4. Signatura instančního konstruktoru se může vyskytovat ve více obměnách, které jsou závislé na počtu a typu dostupných formálních parametrů. Signatura konstruktoru tak může obsahovat jeden, dva, případně libovolný počet formálních parametrů. Prostřednictvím těchto parametrů může konstruktor přijímat datové argumenty, které mu poskytne klientský programový kód. Argumenty mohou být formálním parametrům konstruktoru předávány hodnotou nebo odkazem. Při první alternativě je parametru nabídnuta kopie původního argumentu, zatímco při druhé variantě je do parametru uložen skutečný argument. Například dále uvedený výpis kódu ukazuje, jak lze instančnímu konstruktoru odevzdat řízený ukazatel na hodnotu instance struktury `System::Int16`.

```
// Kód definující __gc třídu je uložen v samostatném
// implementačním souboru jazyka C++ s Managed Extensions.
#include "stdafx.h"
#using <mscorlib.dll>

namespace Pokusy
{
   public __gc class X
   {
   private:
```

```
      System::Int16 i;
   public:
      X(System::Int16 __gc * j) {i = *j;}
      System::Int16 Ziskat_i(){return i;}
   };
}

...

// Praktické použití instančního konstruktoru řízené třídy.
System::Int32 a = 100;
System::Int16 __gc * p_a =
   reinterpret_cast<System::Int16 __gc *>(&a);
*p_a += 100;
Pokusy::X __gc * obj1 = __gc new Pokusy::X(p_a);
MessageBox::Show(obj1->Ziskat_i().ToString());
```

POZNÁMKA



V okně se zprávou bude zobrazena hodnota 200, což je logické, protože pomocí řízeného ukazatele `p_a` uskutečňujeme inkrementaci lokální hodnotové proměnné `a` typu `System::Int32`. Jak již víte, kompilátor jazyka C++ s Managed Extensions provádí implicitní inicializaci všech `__gc` ukazatelů na nulové hodnoty. Každý deklarovaný `__gc` ukazatel je proto okamžitě po svém vytvoření transformovaný na tzv. prázdný ukazatel (tedy ukazatel, který nebyl zatím inicializován referencí na instanci jistého datového typ).

UPOZORNĚNÍ

Jestliže se blíže podíváme na typ `System::Int16`, zjistíme, že za ním ve skutečnosti stojí stejnojmenná hodnotová (`__value`) struktura. Z charakteru práce hodnotových struktur vyplývá, že instance těchto entit jsou standardně alokovány na zásobníku vlákna a ne na řízené hromadě. Jak je potom možné, že můžeme deklarovat řízený ukazatel, který je nasměrován na instanci hodnotové struktury? Jazyková specifikace praví, že `__gc` ukazatel smí být namířen, kromě platné instance `__gc` třídy, také na jakoukoliv platnou instanci hodnotové (`__value`) třídy. To znamená, že navzdory svému primárnímu určení může být `__gc` ukazatel použit také pro přístup k instancím `__value` tříd a struktur. Po pravdě řečeno, kompilátor jazyka C++ s Managed Extensions determinuje typ ukazatele směrujícího na jistou entitu na základě datového typu této entity. Řečeno jinak, je-li touto entitou hodnotový nebo odkazový datový typ definovaný v bázové knihovně tříd vývojově-exekuční platformy .NET Framework 1.1, kompilátor bude implicitně emitovat generování instrukcí, jež vytvoří řízený, tedy `__gc` ukazatel. Na druhé straně, pokud použije programátor nativní typ (kupříkladu `int`, `char` či nativní (`__nogc`) třídu), pak bude kompilátorem sestrojen adekvátní neřízený (`__nogc*`) ukazatel.

5. Jedna řízená třída může definovat několik verzí instančního konstruktoru za předpokladu, že každá definice konstruktoru se odlišuje od ostatních verzí

dotyčného konstruktoru svou signaturou. Jestli je splněna tato podmínka, říkáme, že třída disponuje přetíženým instančním konstruktorem. Signatury přítomných definicí instančního konstruktoru se mohou lišit počtem deklarovaných formálních parametrů, jejich datovými typy a také jejich pořadím. Přetížení instanční konstruktor řízené třídy by mohl vypadat třeba následovně:

```
public __gc class X
{
private:
   System::Int16 i;
public:
   X(){i = 0;}
   X(System::Byte b){i = b;}
   X(System::Char ch, System::Byte b){}
   X(System::Byte b, System::Char ch){}
   X(System::Int16 __gc * j){i = *j;}
   System::Int16 Ziskat_i(){return i;}
};
```

6. Pokud se v těle řízené třídy nenachází definice instančního konstruktoru, kompilátor jazyka C++ s Managed Extensions vygeneruje implicitní veřejně přístupný konstruktor. Podoba sestaveného výchozího konstruktoru v kódu jazyka MSIL je takováto:

```
.method public specialname rtspecialname
        instance void  .ctor() cil managed
{
  // Code size       7 (0x7)
  .maxstack  1
  IL_0000:  ldarg.0
  IL_0001:  call       instance void [mscorlib]System.Object::.ctor()
  IL_0006:  ret
} // end of method X::.ctor
```

MSIL kód přináší pár zajímavých informací. Především, instanční konstruktor je v tomto prostředí reprezentován speciální metodou s názvem `.ctor`, která obsahuje přeložený řízený kód. V těle metody se nacházejí tři programové instrukce jazyka MSIL:

1. Instrukce `IL_0000: ldarg.0` je zodpovědná za umístění ukazatele na instanci třídy (`this`) na zásobník.

2. Instrukce `IL_0001: call instance void [mscorlib]System.Object::.ctor()` aktivuje instanční konstruktor bázové systémové třídy `System::Object`. Jak jsme si již řekli, všechny řízené třídy jsou v jazyce C++ s Managed Extensions buď přímo, anebo nepřímo odvozeny od primární mateřské třídy `Object` z jmenného prostoru `System`. Naše `__gc` třída `X` je proto nepřímým potomkem třídy `System::Object`.

UPOZORNĚNÍ	
	Uvažujeme-li o řetězci tříd, jenž vzniká postupným odvozováním právě jedné třídy od bázové třídy, pak můžeme prohlásit, že pro aktivaci instančních konstruktorů platí následující pravidlo: Třída, nacházející se na konci řetězce dědičnosti, tedy posledně odvozená třída, volá prostřednictvím svého instančního konstruktoru konstruktor své mateřské třídy (tedy předposledně odvozené třídy). Tato mateřská třída, vystupující také jako odvozená třída (nakolik dědí své charakteristiky od jiné třídy v řetězci), zase volá konstruktor své bázové třídy. Tento cyklus pokračuje tak dlouho, dokud není aktivován konstruktor primární bázové třídy stojící na začátku řetězce dědičnosti. Jelikož naše `__gc` třída `X` není podtřídou žádného přímého předka, dochází v těle jejího instančního konstruktoru k aktivaci konstruktoru nepřímé bázové třídy, jíž je třída `System::Object`.

3. Instrukce `IL_0006: ret` má v kompetenci ukončení působení volané metody a navrací kontrolu nad další exekucí aplikace do rukou volající metody.

Abychom si předvedli, jak pracuje implicitní volání instančních konstruktorů mezi jednotlivými třídami v řetězci dědičnosti, uvažujme následující fragment programového kódu:

```
#include "stdafx.h"
#using <mscorlib.dll>
using namespace System::Windows::Forms;

namespace Pokusy
{
   public __gc class X
   {
   public:
      X()
      {
         MessageBox::Show(S"Byl aktivován instanční konstruktor třídy X.");
      }
   };

   public __gc class Y : public X
   {
   public:
      Y()
      {
         MessageBox::Show(S"Byl aktivován instanční konstruktor třídy Y.");
      }
   };

   public __gc class Z : public Y
   {
   public:
      Z()
      {
         MessageBox::Show(S"Byl aktivován instanční konstruktor třídy Z.");
      }
   };
}
```

...

```
// Instanciace řízené třídy Z.
Pokusy::Z __gc * obj_Z = __gc new Pokusy::Z();
```

Kód představuje tři řízené třídy (`X`, `Y`, `Z`), které jsou uloženy ve společném jmenném prostoru `Pokusy`. Všechny tři třídy definují veřejně přístupné instanční konstruktory, přičemž mezi třídami existuje vzájemný vztah vybudovaný na bázi dědičnosti. Třída `X` je nepřímou podtřídou systémové třídy `System::Object`, třída `Y` je přímým potomkem třídy `X` a konečně, třída `Z` vznikla odvozením od třídy `Y`. Ačkoliv jsme se ještě nevěnovali koncepci dědičnosti v prostředí .NET, musíme si v této souvislosti uvést několik relevantních poznatků. Uvnitř společné jazykové infrastruktury platformy .NET Framework 1.1 je možné použít pouze jednoduchou veřejnou dědičnost, v rámci které má každá řízená třída právě jednoho přímého předka. To tedy znamená, že kterákoliv `__gc` třída nemůže dědit své charakteristiky od více tříd současně (vícenásobná dědičnost je ovšem stále přípustná ve spojení s nativními třídami).

Omezení plynoucí ze zavedení jednoduché dědičnosti je možné efektivně obejít, protože řízená třída může implementovat libovolný počet řízených rozhraní (`__gc interface`). Stejně významné je poukázat na skutečnost, že v jazyce C++ s Managed Extensions mohou programátoři využít jenom veřejnou dědičnost – klíčové slovo `public`, které je následováno specifikací bázové třídy, je nutnou součástí deklaračního příkazu odvozené třídy. Pokud se vrátíme zpět k výše zobrazenému kódu, můžeme říci, že v okamžiku, kdy dojde k vytvoření instance `__gc` třídy `Z`, budou kompilátorem aktivovány instanční konstruktory tříd `X`, `Y` a `Z`.

V této souvislosti připomínáme, že kompilátor volá instanční konstruktory tříd `X` a `Y` ještě předtím, než začne realizovat kód zapsaný v instančním konstruktoru třídy `Z`. Jestli nakoukneme do kódu MSIL, shledáme, že kompilátor implicitně aktivuje veřejně přístupné bezparametrické konstruktory. Kdybychom kupříkladu instanční konstruktory tříd `X`, `Y` a `Z` přetížili dle níže uvedeného vzoru, kompilátor by postupoval tak, že nejprve by aktivoval bezparametrické instanční konstruktory tříd `X` a `Y` a vzápětí by spustil kód parametrického instančního konstruktoru třídy `Z`.

```
#include "stdafx.h"
#using <mscorlib.dll>
using namespace System::Windows::Forms;

namespace Pokusy
{
   public __gc class X
   {
   public:
      X()
      {
         MessageBox::Show(String::Concat(S"Byl aktivován instanční ",
         S"konstruktor třídy X."));
      }
      X(System::Int32 x)
      {
         MessageBox::Show(S"X::X");
      }
   };

   public __gc class Y : public X
   {
   public:
```

```
      Y()
      {
         MessageBox::Show(String::Concat(S"Byl aktivován instanční ",
         S"konstruktor třídy Y."));
      }
      Y(System::Int32 y)
      {
         MessageBox::Show(S"Y::Y");
      }
   };

   public __gc class Z : public Y
   {
   public:
      Z()
      {
         MessageBox::Show(String::Concat(S"Byl aktivován instanční ",
         S"konstruktor třídy Z."));
      }
      Z(System::Int32 z)
      {
         MessageBox::Show(S"Z::Z");
      }
   };
...
// Aktivace přetíženého konstruktoru třídy Z.
Pokusy::Z __gc * obj_Z = __gc new Pokusy::Z(10);
```

7. Instanční konstruktor definovaný v řízené třídě nedovede pracovat se standardními formálními parametry. Tímto termínem se označují formální parametry signatury konstruktoru, které jsou inicializovány předem definovanými hodnotami ještě předtím, než kompilátor začne zpracovávat programový kód, jenž se nachází v těle konstruktoru. Kdybyste použili instanční konstruktor se standardním formálním parametrem, kompilátor by vygeneroval chybové hlášení.

   ```
   public __gc class A_1M
   {
   public:
   // Použití standardního formálního parametru v signatuře
   // instančního konstruktoru řízeného třídy
   // jazyk C++ s Managed Extensions nepřipouští.
      A_1M(System::Byte b = 10){}
   };
   ```

   Přestože instanční konstruktor `__gc` třídy si se standardním formálním parametrem nerozumí, při definici neřízené (`__nogc`) třídy je situace úplně jiná. Tu totiž můžete instanční konstruktor obdařit standardním formálním parametrem bez jakýchkoliv potíží.

   ```
   __nogc class A_1
   {
   private:
   int a;
   public:
         A_1(unsigned short h = 12){a = h;}
         unsigned short ZjistitHodnotu(){return a;}
   };

   ...
   ```

```
Pokusy::A_1 __nogc * obj = __nogc new Pokusy::A_1();
// V dialogu bude zobrazena hodnota 12.
MessageBox::Show(obj->ZjistitHodnotu().ToString());
delete obj;
```

8. Instanční konstruktor řízené třídy může inicializovat datové položky instance podle více scénářů:

    - Konstruktor může uvést privátní datové položky do výchozího stavu pomocí jejich explicitní inicializace ve svém těle.

    - Konstruktor může datové položky explicitně inicializovat pomocí speciálního syntaktického zápisu, který je známý jako inicializační seznam.

    - Konstruktor může aktivovat jinou členskou metodu třídy, která převezme zodpovědnost za inicializaci datových položek sestrojené instance třídy.

Statické konstruktory

V dřívějších kapitolách jsme se věnovali podrobné charakteristice instančních konstruktorů, takže je načase, abychom popojeli dál. Náplní této kapitoly tudíž budou konstruktory statické. Hned na úvod se musíme obeznámit s tím, že možnost ukládat definice statických konstruktorů do těl řízených (`__gc`) tříd je jednou z novinek, jež přináší programovací jazyk C++ s Managed Extensions. Statický konstruktor si můžete představit jako členskou metodu třídy, jejíž úlohou je inicializovat její statické datové položky. Statický konstruktor se proto váže k samotné třídě jako celistvému datovému typu a ne k jednotlivým instancím čili objektům tohoto typu. Každá řízená třída smí definovat statický konstruktor. Ten ovšem musí vyhovovat těmto kriteriím:

1. Statický konstruktor musí mít stejný název jako třída, v těle které je umístěn. V tomto směru se tedy statický konstruktor nijak nediferencuje od svého instančního protějšku.

2. Hlavička definičního příkazu statického konstruktoru musí obsahovat modifikátor `static`. Klíčové slovo `static` vytváří z členské metody statický konstruktor.

3. Statický konstruktor musí být bezparametrický, což znamená, že nesmí zavádět deklaraci žádných formálních parametrů.

4. Statický konstruktor nesmí pracovat s žádnou návratovou hodnotou – není dovoleno užít klíčové slovo `void`, ani instanci struktury `System::Void`.

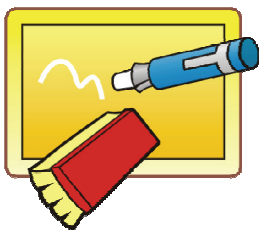
Jelikož primárním účelem statického konstruktoru je inicializace statických datových položek řízené třídy, je nutné, aby k exekuci tohoto typu konstruktoru došlo ještě před vytvořením první instance dotyčné `__gc` třídy. Tato podmínka je splněna, protože ve skutečnosti je statický konstruktor implicitně volán běhovým prostředím CLR okamžitě po načtení programového kódu řízené třídy do operační paměti počítače. Bude dobré, když si zapamatujete, že statický konstruktor je prostředím CLR aktivován právě jedenkrát. To je pochopitelné, neboť statické datové položky je nutno inicializovat pouze po načtení řízené třídy a ne až při zakládání nových instancí této třídy. Po zpracování statického konstruktoru budou všechny statické datové položky řízené třídy uvedeny do výchozího stavu, což znamená, že budou moci být využitelné instancemi `__gc` třídy, které budou založeny v následujících časových okamžicích.

První programová ukázka, kterou si představíme, demonstruje použití statického konstruktoru v řízené třídě s názvem `Aplikace`. Seznamte se nejdřív s předmětným zdrojovým kódem, a pak si k němu řekneme pár slov.

```
#include "stdafx.h"
#using <mscorlib.dll>
using namespace System;
namespace Pokusy
{
   __gc class Aplikace
   {
   private:
      static String __gc * JmenoAplikace;
      static Aplikace()
      {
         JmenoAplikace = S"notepad";
      }
   public:
      static String __gc * ZobrazitJmenoAplikace()
      {
         return JmenoAplikace;
      }
      static Void SpustitAplikaci(String __gc * CestaKSouboruAplikace)
      {
         System::Diagnostics::Process __gc * NovyProces =
            __gc new System::Diagnostics::Process();
         if (CestaKSouboruAplikace->Length != 0)
            NovyProces->Start(CestaKSouboruAplikace);
         else
            NovyProces->Start(JmenoAplikace);
      }
   };
}
```

Jak můžete pozorovat, řízená třída `Aplikace` definuje statický konstruktor, který uskutečňuje inicializaci soukromé statické datové položky třídy s názvem `JmenoAplikace` typu `String __gc*` (řízený ukazatel na instanci třídy `String` z jmenného prostoru `System`). Všimněte si syntaktickou formu statického konstruktoru: Před názvem konstruktoru stojí modifikátor `static`, který nám zřetelně dává na vědomí, že pracujeme se statickým konstruktorem. Za názvem konstruktoru jsou situovány prázdné závorky – to je v pořádku, protože statický konstruktor musí být bezparametrický. V těle konstruktoru je do statické odkazové proměnné `JmenoAplikace` uložen textový řetězec určující název aplikace.

POZNÁMKA



Zajímavostí je, že statický konstruktor může být definován v privátní části řízené třídy bez toho, aby byla jakkoliv narušena jeho správná činnost. Když tuto situaci porovnáme s instančním konstruktorem, zjistíme, že v případě uložení kódu instančního konstruktoru do soukromé datové sekce řízené třídy bychom nebyli schopni vytvořit ani jednu instanci této třídy. Je to proto, že instanční konstruktor by byl privátní, takže klientský kód by k němu nemohl přistupovat.

Statický konstruktor může pracovat pouze se statickými datovými položkami, které přináleží řízené třídě. Z uvedeného důvodu není možné, aby byl statický konstruktor

aplikován ve spojení s inicializací nestatických, respektive instančních datových položek. Je to konec konců srozumitelné: Jelikož je statický konstruktor aktivován běhovým prostředím CLR mnohem dříve, než vůbec může být vytvořena první instance `__gc` třídy, není jednoduše možné, aby tento konstruktor realizoval inicializaci datových položek instancí řízené třídy.

Statické datové položky řízených tříd mohou být explicitně inicializovány. Kdybychom kupříkladu chtěli odkazovou proměnnou `JmenoAplikace` typu `String __gc*` inicializovat textovým řetězcem, mohli bychom tak učinit následovně:

```
static String __gc * JmenoAplikace = S"mspaint";
```

Naproti tomu, v prostředí nativních (`__nogc`) tříd je možné explicitně inicializovat pouze statické konstanty integrálních datových typů:

```
__nogc class A
{
private:
   // Zatímco explicitní inicializace statické konstanty
   // celočíselného datového typu int je v nativním C++ povolena...
   const static int x = 10;

   // ...explicitní inicializaci statické proměnné
   // neintegrálního datového typu není možné uskutečnit.
   static bool y = false; // Zde bude generována chybová výjimka.
public:
   int Ziskat_x(){return x;}
};
```

V okamžiku, kdy kompilátor odhalí náš pokus o explicitní inicializaci statické proměnné `y` typu `bool`, budeme upozorněni na nemožnost realizace zamýšlené operace.

V těle naší řízené třídy jsou uloženy dvě statické metody: `ZobrazitJmenoAplikace` a `SpustitAplikaci`. Prostřednictvím první metody můžeme získat název aplikace v podobě textového řetězce. Kód druhé metody je o něco komplikovanější, protože uskutečňuje instanciaci třídy `Process` z jmenného prostoru `System::Diagnostics`, jejíž pomocí budeme iniciovat spuštění specifikované aplikace. Metoda pracuje s jedním formálním parametrem `CestaKSouboruAplikace` typu `String __gc*`. Pokud programátor uloží do tohoto parametru cestu ke spustitelnému (.exe) souboru aplikace, metoda zabezpečí nastartování zvolené aplikace. Bude-li ovšem parametru odevzdán prázdný řízený textový řetězec (`S""`), bude vyvolaná implicitně určená aplikace, kterou determinuje statický konstruktor třídy. Působení statického konstruktoru `__gc` třídy `Aplikace` můžeme otestovat jediným řádkem kódu jazyka C++ s Managed Extensions:

```
Pokusy::Aplikace::SpustitAplikaci(S"");
```

Důsledkem práce tohoto kódu bude spuštění aplikace Poznámkový blok.

Statický konstruktor libovolné řízené třídy je v kódu jazyka MSIL reprezentovaný metodou s názvem `.cctor` (tento název implikuje zkratku anglického slovního spojení „class constructor“). Použijeme-li nástroj IL DASM a nahlédneme-li na sestavený MSIL kód statického konstruktoru `__gc` třídy `Aplikace`, spatříme tyto programové instrukce:

```
.method private specialname rtspecialname static
        void  .cctor() cil managed
{
  // Code size       11 (0xb)
  .maxstack  1
```

```
  IL_0000:  ldstr      "notepad"
  IL_0005:  stsfld     string Pokusy.Aplikace::JmenoAplikace
  IL_000a:  ret
} // end of method Aplikace::.cctor
```

Statický konstruktor `__gc` třídy má v kódu jazyka MSIL také další pojmenování, z nichž se nejčastěji zmiňují výrazy „konstruktor třídy" a „typový inicializátor". Statický konstruktor je ztělesňován metodou `.cctor`, přičemž tento název představuje klíčové slovo jazyka MSIL, které je vyhrazeno pro identifikaci typového inicializátoru (nejedná se tedy o běžný název dle standardních pravidel pro pojmenování metod). Při bližším pohledu na hlavičku konstruktoru třídy identifikujeme několik příznaků:

1. `private`: Příznak říká, že metoda je přístupná pouze v rámci datového typu, v němž je definována, anebo v rámci typu, jenž je vnořen do typu, ve kterém je tato metoda definována.

2. `specialname`: Příznak determinuje speciální postavení metody v soustavě metod, jež jsou definovány v daném datovém typu.

3. `rtsspecialname`: Tento příznak deklaruje speciální pojmenování metody, které interně používá společné běhové prostředí CLR platformy .NET Framework. Příznak `rtsspecialname` se musí objevovat bok po boku s příznakem `specialname`. Nástroj IL DASM příznak `rtsspecialname` zobrazuje pouze pro informaci, protože IL Assembler použití tohoto příznaku ignoruje.

4. `static`: Příznak připomíná skutečnost, že metoda je statická, tedy že je sdílena všemi instancemi zvoleného datového typu.

5. `void`: Klíčové slovo `void` jazyka MSIL explicitně specifikuje, že metoda nevrací žádnou hodnotu. Jenom pro zajímavost uveďme, že zatímco C++ s Managed Extensions použití klíčového slova `void` v hlavičce statického konstruktoru výslovně zakazuje, jazyk MSIL naopak přítomnost tohoto klíčového slova ve spojení s konstruktorem třídy vyžaduje.

6. Poslední dva příznaky, `cil` a `managed`, říkají, že programové instrukce typového inicializátoru `.cctor` spadají pod řízený kód, jenž je implementován v jazyce MSIL.

Celková kapacitní náročnost konstruktoru třídy s názvem `.cctor` je v jazyce MSIL 11 bajtů (B). V těle konstruktoru se nacházejí tři programové instrukce označené příslušnými návěstími:

1. Instrukce `IL_0000: ldstr "notepad"` na základě textového řetězce znakové sady Unicode vytváří instanci třídy `[mscorlib]System.String` a objektovou referenci na sestrojenou instanci ukládá na zásobník.

2. Instrukce `IL_0005: stsfld string Pokusy.Aplikace::JmenoAplikace` umísťuje objektovou referenci uloženou na zásobníku do statické datové položky třídy.

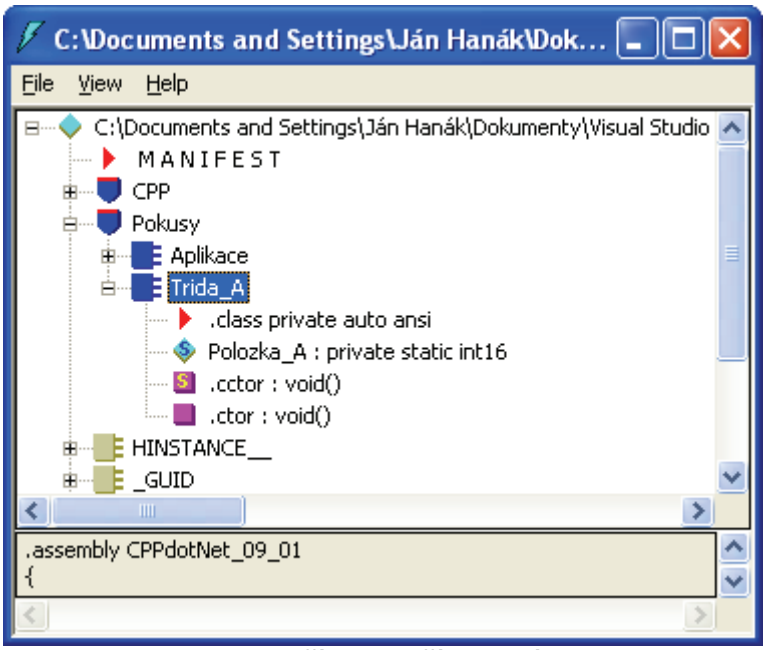
3. Instrukce `IL_000a: ret` ukončuje působení typového inicializátoru a vrací řízení nad během exekuce do rukou řízené aplikace.

Již jsme si vysvětlili, že v prostředí řízených tříd mohou programátoři uskutečňovat explicitní inicializaci statických datových položek. Nyní se však pokusme podívat na to, jak tento proces probíhá. Předpokládejme, že máme k dispozici řízenou třídu s názvem

`Trida_A`, v soukromé části které dochází k explicitní inicializaci statické hodnotové proměnné `Polozka_A` typu `System::Int16`.

```
__gc class Trida_A
{
   static System::Int16 Polozka_A = 100;
};
```

Domníváme se, že budete překvapeni, když vyhlásíme, že explicitní inicializace statické proměnné se uskutečňuje v implicitně sestaveném statickém konstruktoru. Nicméně ano, je to doopravdy tak. Kompilátor jazyka C++ s Managed Extensions při požadavku na uložení celočíselné hodnoty do předem připravené statické proměnné iniciuje sestrojení statického konstruktoru. Tento proces se odehrává „na pozadí", a proto není z prostředí editoru zdrojového kódu vývojářského nástroje Visual C++ .NET 2003 viditelný. Abychom záhadu rozlouskli, musíme se opět ponořit do hlubin symbolických instrukcí jazyka MSIL. Spusťme tedy nástroj IL DASM a prozkoumejme součásti `__gc` třídy `Trida_A`.

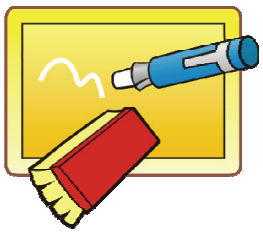


Obr. 1.4: Pohled na řízenou třídu v nástroji IL DASM

Z obr. 1.4 je zřejmé, že kompilátor sestavil kromě statické datové položky třídy (`Polozka_A`) rovněž implicitní statický konstruktor (`.cctor`) a dokonce také implicitní instanční konstruktor (`.ctor`). V těle implicitního statického konstruktoru se nacházejí programové instrukce, které zabezpečují explicitní inicializaci statické datové položky třídy:

```
.method public specialname rtspecialname static
        void  .cctor() cil managed
{
  // Code size       8 (0x8)
  .maxstack  1
  IL_0000:  ldc.i4.s   100
  IL_0002:  stsfld     int16 Pokusy.Trida_A::Polozka_A
  IL_0007:  ret
} // end of method Trida_A::.cctor
```

V tomto fragmentu MSIL kódu můžeme vidět instrukci `ldc.i4.s 100`, která umísťuje na zásobník určenou celočíselnou hodnotu.

POZNÁMKA	
	Programová instrukce `ldc` (což je zkratka z anglického *load numeric constant*) slouží k načítání a uložení determinované hodnoty na zásobník. Generická podoba této instrukce je `ldc.Kapacita[.s]Hodnota`, přičemž identifikátor `Kapacita` představuje alokační kapacitu identifikátoru `Hodnota` v bajtech. Volitelný identifikátor `[.s]` reprezentuje speciální zkrácenou formu kódování pro 4bajtové celočíselné hodnoty z intervalu <-128, 127>. Identifikátor `Hodnota` charakterizuje konkrétní hodnotu, která má být na zásobník načtena. Hodnoty, s nimiž instrukce `ldc` pracuje, jsou definovány prostřednictvím čtyř typů alokačních kapacit, jež jsou mapovány na odpovídající identifikátory: `I4` (4bajtové celočíselné hodnoty), `I8` (8bajtové celočíselné hodnoty), `R4` (4bajtové reálné hodnoty) a `R8` (8bajtové reálné hodnoty). V našem případě má instrukce podobu `ldc.i4.s 100`, což znamená, že na zásobník bude načtena celočíselní hodnota o kapacitě 4 bajty typu `int32`[1], přičemž prostřednictvím identifikátoru `.s` je aplikována efektivnější realizace této operace (je to proto, že zvolená hodnota 100 patří do definičního oboru intervalu <-128, 127>). Po načtení inicializační hodnoty 100 na zásobník je aktivována instrukce `stsfld int16 Pokusy.Trida_A::Polozka_A`, která ukládá tuto hodnotu do statické proměnné `Polozka_A` typu `int16`.

Řízené destruktory, finalizační metody a správa objektů

Poté, co jsme našli cestu z končin, jež obývají řízené konstruktory (tak instanční jakožto i statické), máme před sebou další dobrodružnou pouť, která nás tentokrát zavede mezi programové prostředky, které jsou nápomocny při řízení poslední etapy životních cyklů objektů v prostředí .NET. Začneme srovnáním, v němž si představíme, jak se k likvidaci objektů přistupovalo v nativním C++ a jak je tato činnost realizována v jazyce C++ s Managed Extensions.

Likvidace objektů z pohledu nativního C++

Pakliže se rozhodneme analyzovat proces likvidace objektů v tradičním a řízeném C++, velice rychle dojdeme k poznání, že fáze dealokace objektových zdrojů byla v nativním C++ mnohem přímočařejší, a podle mnoha programátorů také jednodušší. Prvním významným pozitivem bylo, že vývojář věděl přesně určit, kdy bude provedena destrukce kýženého objektu. Pokud bylo zapotřebí před samotnou likvidací objektu uskutečnit některé akce, programátor mohl to těla nativní třídy vložit kód pro destruktor. Destruktor pracoval jako „pohotovostní" metoda, která byla volána těsně před uvolněním objektu z nativní hromady. Když budeme vycházet z toho, že nativní objekt byl zkonstruován dynamicky pomocí operátoru `new`, programátor mohl při nutnosti ukončení jeho činnosti sáhnout po operátoru `delete`, který odstartoval proces destrukce dotyčného objektu. Jinak řečeno, před dealokací objektu byl aktivován destruktor, který vykonal „údržbu"

[1] Na tomto místě je `int32` datovým typem jazyka MSIL a nikoliv jazyka C++ s Managed Extensions.

zdrojů a hned nato byl objekt zlikvidován (pokud nativní třída, z níž byl objekt vytvořen, dědila své charakteristiky z jiné třídy, případně tříd, po aktivaci destruktoru objektu byly zavolány také destruktory tříd v hierarchii dědičnosti).

Vysoce ceněnou vlastností tradičního procesu likvidace objektů v jazyce C++ byla exaktnost a determinovatelnost realizace jeho jednotlivých částí. Jestli programátor použil operátor `delete`, mohl si být jist, že proces likvidace objektu, jenž byl spojen s explicitní exekucí kódu destruktoru, byl okamžitě nastartován. Samozřejmě, likvidace objektů se mohla odehrávat i podle jiných scénářů, nemusela byt nutně iniciována výsostně použitím operátoru `delete`. Tento operátor se používal jako „doplněk" operátoru `new`: pokud byl objekt založen pomocí operátoru `new`, bylo jej nutno explicitně odstranit operátorem `delete`, jinak hrozil vznik nepříznivých situací, o nichž si za chvíli povíme víc. Jestliže byl objekt alokován na zásobníku, což byla v nativním C++ celkem běžná situace, jeho destruktor byl zavolán automaticky ve chvíli, kdy se příslušná lokální automatická proměnná dostala mimo svůj obor platnosti. Nakonec, destruktory objektů mohly být vyvolávány též v rámci ukončení exekuce kódu nativní aplikace.

Abychom výklad zbytečně nekomplikovali, přijmeme dohodu, že v našich úvahách budeme zkoumat pouze životní cykly objektů sestrojovaných operátorem `new` a odstraňovaných operátorem `delete`. Bohužel, tento proces paměťového managementu se pojil s nemálo temnými stránkami:

1. Velice často se stávalo, že programátor zapomněl použít operátor `delete` a uvolnit tak objekt z nativní hromady. Proto docházelo k neslavně proslulým programátorským chybám, jež vyústily do vzniku paměťových děr. Paměťová díra identifikuje část operační paměti počítače, která byla neefektivně alokována již nepoužívaným, anebo nepoužitelným objektem. Poněvadž objekt nebyl korektně zlikvidován, stále okupoval cenné paměťové zdroje, čímž došlo nejenom k nehospodárnému využívání dostupných alokovatelných paměťových jednotek, ale nezřídka byla snížena také výkonnost samotné aplikace. Přešlap programátora, jehož důsledkem byl vznik paměťových děr, byl o to kritičtější, o co větší paměťový segment byl nadarmo obsazen.

2. Přestože paměťové díry byly nepochybně velikým strašákem, snad ještě větším prohřeškem byly pokusy o opětovnou likvidaci jednou odstraněného objektu z nativní hromady. Tento problém se stupňoval zejména v případě kruhových referencí, kdy jeden objekt držel referenci na druhý objekt a druhý zas na ten první. Problémem bylo určit, který objekt je zodpovědný za dealokaci zdrojů (ať už svých nebo kolegových) a chyby vznikaly jako na běžícím páse.

Z uvedených informací plyne, že tradiční paměťový management známý z nativního C++ byl ve skutečnosti manuálním procesem: všechny akce spojené s destrukcí objektu zůstávaly v působnosti programátora a jedině ten byl odpovědný za jejich správnou realizaci. Na druhé straně, celý proces byl relativně snadno predikovatelný, takže vývojáři přesně věděli, jaké operace jsou právě prováděné.

Likvidace objektů z pohledu řízeného C++

Vývojově-exekuční platforma Microsoft .NET Framework s sebou přinesla společné běhové prostředí CLR, které poskytuje řízeným aplikacím mnohé služby nízké úrovně, správou programových vláken počínaje a automatickou kontrolu objektů na řízené hromadě konče. Ano, jedním z lákadel nového prostředí je nepochybně automatický správce paměti, jenž oprošťuje programátory od používání operátoru `delete` a explicitní dealokace řízených objektů. To je všechno pravda, no nahlédneme-li do problematiky

přece jenom o něco více, zjistíme, že mnohé věci jsou mnohem komplikovanější, než se na první pohled jeví.

V jazyce C++ s Managed Extensions se řízené objekty vytvářejí pomocí operátoru `__gc new`. Každý objekt může být vybaven svým destruktorem, který však ve skutečnosti není přímým ekvivalentem destruktoru z nativního C++. Destruktor v těle řízených tříd je totiž implicitně konvertovaný do podoby finalizační metody `Finalize`, která je aktivována automatickým správce paměti v určitém okamžiku před uvolněním řízeného objektu z řízené hromady. Problém je, že tento okamžik není možné přesně určit, a proto se vývojáři nemohou spolehnout na to, že kód finalizační metody bude zpracován v předem vymezeném časovém okamžiku (jde o takzvanou nedeterministickou finalizaci řízených objektů).

Příslušná finalizační metoda bude podrobena exekuci v procese kolekce té generace řízené hromady, v níž je daný objekt umístěn. Automatický správce paměti při své činnosti označí ty objekty, které si vyžadují aktivaci svých finalizačních metod – tyto objekty budou následně uloženy do seznamu a k jejich finalizaci bude docházet na samostatném finalizačním vláknu.

TIP

S nedeterministickou finalizací se můžeme poprat: Ačkoliv okamih zpracování finalizační metody není programátorovi znám, je jisté, že tato metoda bude vykonána v procese kolekce generací řízené hromady. Mohlo by se tedy zdát, že když nalezneme způsob, jak explicitně podnítit nastartování automatické správy paměti v jisté objektové generaci, máme vyhráno. Takováto možnost doopravdy existuje – pokud zavoláme statickou metodu `Collect` třídy `GC` z jmenného prostoru `System`, bude povolán automatický správce paměti, který nařídí realizaci analýzy buď celé řízené hromady (je-li metoda `Collect` zavolána bez dodání vstupních argumentů), anebo zvolené generace řízené hromady (pokud metodě `Collect` předáme číselnou identifikaci té generace, která má být prozkoumána). Pro zjištění generace, v níž je objekt situován, můžeme použít statickou metodu `GetGeneration` třídy `GC`:

```
Button __gc * b = __gc new Button();
MessageBox::Show(GC::GetGeneration(b).ToString());
```

Přestože je automatický správce paměti schopen uvolnit nepotřebné objekty, je nutno připomenout, že jde pouze o řízené nepotřebné objekty. To znamená, že jestli ve své aplikaci pracujete s nativními prostředky operačního systému, případně s jinými neřízenými zdroji, je bezpodmínečně nevyhnutné, abyste tyto zdroje explicitně dealokovali sami, nakolik automatický správce paměti nesleduje efektivnost využívání tohoto typu systémových zdrojů. Uvolnit nativní prostředky můžete v podstatě kdykoliv, když je již nebudete potřebovat – v tomto směru je velice užitečná metoda `Dispose` rozhraní `IDisposable`, která umožňuje dealokovat neřízené zdroje deterministicky, tedy na požádání. Naneštěstí, metoda `Dispose` musí být zavolána klientským kódem, na což mnozí programátoři zapomínají.

Řízený destruktor

Abychom se vyhnuli terminologickým nepřesnostem, budeme pro destruktor řízených tříd jazyka C++ s Managed Extensions používat termín řízený destruktor. Jak již bylo vzpomenuto, tento destruktor není ryzím destruktorem, nýbrž finalizační metodou, která je v procesu nedeterministické finalizace volána automatickým správcem paměti. Finalizační metodu `Finalize` ovšem do těla `__gc` třídy vložit nesmíme: za tímto účelem je nutno použít syntaxi řízeného destruktoru.

Řízený destruktor se velice ponáší na standardní destruktor nativních tříd: má stejný název jako `__gc` třída, před jeho názvem se nachází symbol vlnovky (`~`), destruktor nepracuje s žádnou návratovou hodnotou a je bezparametrický. Podívejme se nyní na podobu řízeného destruktoru v ukázkové řízené třídě:

```
#include "stdafx.h"
#using <mscorlib.dll>
#using <System.Windows.Forms.dll>
using namespace System::Windows::Forms;
namespace Pokusy
{
__gc class A
{
public:
   // Instanční konstruktor.
   A()
   {
      MessageBox::Show(S"Instanční konstruktor třídy A.",
      S"Řízený konstruktor", MessageBoxButtons::OK,
      MessageBoxIcon::Information);
   }
   // Řízený destruktor.
   ~A()
   {
      MessageBox::Show(S"Řízený destruktor třídy A.",
      S"Řízený destruktor", MessageBoxButtons::OK,
      MessageBoxIcon::Information);
   }
};
}
```

Prohlédneme-li si pomocí aplikace IL DASM programový kód `__gc` třídy `A` v jazyce MSIL, dozvíme se, že řízený destruktor je v tomto prostředí reprezentovaný metodou `__dtor`.

```
.method public instance void  __dtor() cil managed
{
  // Code size       13 (0xd)
  .maxstack  1
  IL_0000:  ldarg.0
  IL_0001:  call       void [mscorlib]System.GC::SuppressFinalize(object)
  IL_0006:  ldarg.0
  IL_0007:  callvirt   instance void Pokusy.A::Finalize()
  IL_000c:  ret
} // end of method A::__dtor
```

Jak si můžete všimnout, destruktor `__dtor` neobsahuje kód pro volání statické metody `Show` třídy `MessageBox`, která je zodpovědná za zobrazení dialogového okna se zprávou (za chvíli si povíme, že tento kód je umístěn ve finalizační metodě `Finalize`). Místo toho jsou v těle řízeného destruktoru volány dvě metody:

1. Statická metoda `SuppressFinalize` třídy `GC`, která zabraňuje realizaci jiných finalizačních akcí pro specifikovaný objekt. Potlačení finalizace má význam při implementaci metody `Dispose`, aby nedošlo k opětovnému uvolnění již jednou dealokovaných systémových zdrojů.

2. Virtuální metoda `Finalize`, která obsahuje kód pro zobrazení dialogu s informační zprávou. Ve všeobecnosti lze říci, že do těla finalizační metody bude vložen jakýkoliv programový kód, jenž zapíšete do řízeného destruktoru při programování v jazyce C++ s Managed Extensions.

Používání finalizačních metod ve spojení s řízenými objekty je kritickou oblastí z hlediska výkonnosti aplikace. Každý objekt, jenž disponuje metodou `Finalize`, je automatickým správcem paměti identifikován a přesunut na finalizační programové vlákno, na kterém budou postupně volány finalizační metody jednotlivých objektů (pořadí volání metod není deterministické, a proto jej nelze přesně determinovat).

Z uvedeného vyplývá, že finalizační metody byste měli využívat pouze v opodstatněných případech. Pokud ve své aplikaci pracujete pouze s řízenými zdroji, není potřebné, abyste své třídy rozšiřovali o řízené destruktory. V případě použití nativních zdrojů přichází implementace řízených destruktorů do úvahy, no pravděpodobně lepším řešením je využití deterministické dealokace prostřednictvím metody `Dispose` rozhraní `IDisposable`. Tuhle metodu můžeme naprogramovat tak, aby na požádání uskutečnila explicitní uvolnění obsazených zdrojů. Aby však metoda `Dispose` mohla odvést svou práci, musí být aktivována přímo klientským kódem. Poněvadž programátoři často zapomínají příslušné metody `Dispose` volat, je nutno vytvořit flexibilní přepojení mezi metodami `Dispose` a odpovídajícími finalizačními metodami čili řízenými destruktory. Když programátor explicitně uvolní zdroje objektu zavoláním metody `Dispose`, finalizační metoda musí být schopna tuto skutečnost zjistit a zabránit pokusu o opětovnou dealokaci již odstraněných zdrojů.

Deterministická finalizace objektů pomocí metody Dispose rozhraní IDisposable

V předcházejících kapitolách jsme si poměrně podrobně představili základní etapy životního cyklu běžného objektu, jenž operuje na řízené hromadě běhového prostředí CLR. Stejně tak víte, že řízená hromada je pod dohledem automatického správce paměti, který analyzuje dostupné objekty a na základě stromu objektových referencí přijímá rozhodnutí o tom, zda bude ten-který objekt z řízené hromady odstraněn nebo ne. Na první pohled by se mohlo zdát, že všechny činnosti spojené s dealokací objektových zdrojů vykoná správce paměti sám, a tedy že v tomto směru není zapotřebí žádný zásah ze strany programátora. Ano, vskutku pravdivé je tvrzení, že správce paměti si poradí s implicitní destrukcí instancí řízených tříd a jejich prostředků. Nicméně v okamžiku, kdy ve své aplikaci začnete používat neřízené čili nativní systémové zdroje, musíte zabezpečit jejich uvolnění sami, neboť tento typ zdrojů se nachází mimo sféry vlivu automatického správce paměti.

Dealokaci nativních zdrojů můžeme uskutečnit v těle řízeného destruktoru, jenž vystupuje jako přestrojená finalizační metoda `Finalize`. Máte pravdu, tento postup je zcela legální, ovšem nevyhýbá se několika problémům. Tak především, všechny objekty, které definují své řízené destruktory (a tak dávají běhovému prostředí CLR na známost nutnost své finalizace), budou muset být zpracovávány na samostatném finalizačním vláknu ve dvojetapovém procesu. Jelikož je nutné explicitně podrobit exekuci finalizační programový kód, destrukce objektů požadujících finalizaci je ve srovnání s objekty, které tyto úkony neiniciují, komplikovanější a výkonově náročnější. Bohužel, tato nevýhoda

nepředstavuje největší „temnou" stránku finalizačního procesu. Mnohem horší je to, že implicitní finalizace je nedeterministická, což je způsobeno tím, že časový okamžik aktivace finalizační metody je v rukou automatického správce paměti a nikoliv programátora. Právě uvedené časová nejednoznačnost při finalizaci řízených objektů je pro nemalý počet programátorů přicházejících z čistého C++ tvrdým oříškem, s nímž se musí vypořádat, jakmile se dostanou do nového programovacího prostředí.

Situace se vyhrocuje zvláště v těch okamžicích, kdy je nutné alokované zdroje uvolnit v přesně stanovený časový okamžik. Jedním z možných řešení je vytvoření speciálních dealokačních metod, jež zajistí uvolnění drahocenných systémových zdrojů v případě potřeby na požádání. Zkonstruovat metody podobného typu není pro vývojáře ve většině případů problémem, ovšem lepším řešením je nabídnout standardizovanou cestu, po které je možné kráčet. Touto pomyslnou cestou je v prostředí vývojově-exekuční platformy .NET Framework 1.1 metoda `Dispose` rozhraní `IDisposable` z jmenného prostoru `System`.

Rozhraní `IDisposable` je řízené (`__gc`), ale přitom velice jednoduché rozhraní, které obsahuje deklaraci prototypu pouze jedné jediné metody, a tou je `Dispose`. Primárním účelem metody `Dispose` je zabezpečit uvolnění řízených a nativních zdrojů, které byly alokovány objektem řízené třídy. Důležité je uvědomit si, že metoda `Dispose` zahajuje proces likvidace obsazených zdrojů na požádání, čímž ve skutečnosti uplatňuje deterministický styl finalizace. Pokud vytvoříte `__gc` třídu a uskutečníte implementaci rozhraní `IDisposable`, budete muset definovat virtuální metodu `Dispose`, v jejímž těle bude umístěn kód pro dealokaci objektových prostředků. Metodu `Dispose` mohou volat jiní programátoři, kteří používají vaši řízenou třídu, a to v okamžiku, kdy již nebudou potřebovat víc pracovat s objektem, respektive tehdy, kdy již nebude zapotřebí spravovat vyhrazené zdroje.

Je zřejmé, že metoda `Dispose` rozhraní `IDisposable` může, no nemusí být explicitně aktivována programátorem. To tedy znamená, že vaše třída (a její instance) se nemůžou stoprocentně spoléhat na to, že metoda `Dispose` bude doopravdy zavolána a že kýžené prostředky budou uvolněny. Pro vyřešení této zapeklité situace je potřebné do těla řízené třídy vložit rovněž definici řízeného destruktoru, který bude působit coby finalizační metoda. Neuskuteční-li programátor explicitní dealokaci nativních objektových zdrojů pomocí metody `Dispose`, převezme realizaci tohoto úkolu na svá bedra automatický správce paměti, který v procesu zpracování objektů na finalizačním vláknu aktivuje příslušnou finalizační metodu instance naší třídy.

Možná se po přečtení předešlých řádků cítíte být poněkud zmateni. Nemějte však strach: abychom předešli jakýmkoliv nejasnostem, shrneme si vzájemné působení metody `IDisposable::Dispose` a řízeného destruktoru do následujícího seznamu:

1. Metodu `Dispose` jakožto i řízený destruktor je v zásadě nutné vytvářet pouze tehdy, pokud vaše `__gc` třída pracuje s nativními systémovými zdroji. K takovýmto zdrojům patří kupříkladu manipulátory oken (handle), manipulátory souborů a databázových spojení, ale také instance nativních (`__nogc`) tříd, jež byly vytvořeny na neřízené C++ hromadě. Protože automatický správce paměti hlídkuje jenom na řízené hromadě, nemá vůbec tušení o nativních prostředcích, které byly alokovány v jiných segmentech operační paměti. Je proto nutné, abyste uvedené zdroje odstranili sami.

2. Dealokaci nativních zdrojů můžete uskutečnit dvěma způsoby, a to buď implicitně, nebo explicitně. Netrváte-li na tom, aby byly nativní zdroje zlikvidovány v jistém časovém okamžiku, můžete použít implicitní dealokační model nativních zdrojů. Podstata tohoto modelu spočívá ve vytvoření řízeného destruktoru, do jehož těla vložíte programový kód pro likvidaci nativních zdrojů. Zapsané instrukce

destruktoru pak budou kompilátorem jazyka C++ s Managed Extensions přetransformovány do podoby finalizační metody `Finalize`. Tato metoda bude aktivována automatickým správcem paměti v procesu finalizace řízené instance na řízené hromadě běhového prostředí CLR. Na druhé straně, pokud je pro vás životně důležité uvolnit obsazené neřízené zdroje v jistém, plně determinovatelném časovém okamžiku, vaším favoritem se zcela určitě stane explicitní dealokační model. Srdcem tohoto modelu je implementace rozhraní `IDisposable` a zavedení definice pro metodu `Dispose`. V těle metody `Dispose` můžete uvolnit tak řízené jako i nativní zdroje, které se staly nepotřebnými. Mějte ovšem dobře na paměti, že metodu `Dispose`, kterou vaše řízená třída definuje, je nutno explicitně vyvolat. Jestli programátor metodu zavolá, je všechno v naprostém pořádku. No pokud vývojář na aktivaci metody `Dispose` zapomene (což se stává velice často), musíme mít k dispozici jakousi „záchrannou brzdu", která nám umožní nativní zdroje správně dealokovat. Roli záchrance v tomto případě sehrává řízený destruktor, respektive adekvátní finalizační metoda. Zjistí-li instance vaší `__gc` třídy, že nedošlo k vyvolání metody `Dispose`, uskuteční destrukci nativních zdrojů ve své finalizační metodě. Jak jsme si již řekli, tato metoda bude aktivována automatickým správcem paměti při likvidaci objektů na řízené hromadě.

3. Mezi metodou `Dispose` a řízeným destruktorem existuje vzájemná vazba, na kterou je dobré myslet zejména v okamžicích, kdy se chystáme pracovat s neřízenými prostředky. Ačkoliv jsme dosud vzpomínali hlavně nativní zdroje, metodu `Dispose` můžeme použít rovněž v souvislosti s řízenými zdroji. Uveďme si příklad. Povězme, že vaše `__gc` třída ve svém instančním konstruktoru vytváří několik řízených grafických objektů, které používá na vykouzlení působivých efektů. V momentě, kdy už založené objekty nejsou potřebné, je můžete v těle metody `Dispose` dealokovat a zabezpečit tak snížení zatížení řízené hromady. Je sice pravda, že řízené zdroje budou dříve či později uvolněné automatickým správcem paměti, no jejich explicitní dealokace je komfortnějším řešením.

Abychom si proces použití metody `Dispose` společně s řízeným destruktorem předvedli v praxi, předpokládejme, že jsme naprogramovali `__gc` třídu `A`, která vypadá následovně:

```
#include "stdafx.h"
#using <mscorlib.dll>
#using <System.dll>

using namespace System;
using namespace System::Windows::Forms;
namespace GC_Tridy
{

public __gc class A : public System::IDisposable
{
private:
   bool bDisposed;
   typedef void * HWND;
   System::Windows::Forms::Form __gc * f;
   HWND Manipulator;
   [System::Runtime::InteropServices::DllImportAttribute
   (S"user32.dll",
   System::Runtime::InteropServices::CharSet = CharSet::Auto)]
   static HWND GetDesktopWindow();
public:
   A()
   {
      bDisposed = false;
```

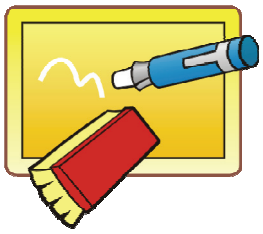
```
      f = __gc new System::Windows::Forms::Form();
      Manipulator = GetDesktopWindow();
   }
   String __gc * Ziskat_Manipulator()
   {
      if (bDisposed)
         throw new System::ObjectDisposedException(this->ToString());
      System::IntPtr m = Manipulator;
      return m.ToString();
   }
   void Dispose()
   {
      Dispose(true);
      GC::SuppressFinalize(this);
   }
   protected: virtual void Dispose(bool Disposing)
   {
      if (Disposing)
      {
         f->Close();
         Manipulator = 0;
         MessageBox::Show(String::Concat(
            S"Metoda Dispose byla aktivována programátorem.",
            System::Environment::NewLine,
            S"Byly uvolněny řízené i nativní zdroje."),
            S"Dealokace zdrojů", MessageBoxButtons::OK,
            MessageBoxIcon::Information);
      }
      else
      {
         Manipulator = 0;
         MessageBox::Show(String::Concat(
         S"Metoda Dispose byla aktivována prostředím CLR.",
         System::Environment::NewLine,
         S"Byly uvolněny nativní zdroje."),
         S"Dealokace zdrojů", MessageBoxButtons::OK,
         MessageBoxIcon::Information);
      }
      bDisposed = true;
   }
   ~A()
   {
      Dispose(false);
   }
};
}
```

Řízená třída `A` implementuje rozhraní `System::IDisposable`, přičemž syntaxe pro uskutečnění implementační operace je stejná jako v případě vytváření odvozených tříd. (I když ve spojení s `__gc` třídami mohou programátoři používat pouze jednoduchou dědičnost, každá řízená třídy smí implementovat libovolný počet `__gc` rozhraní.)

V soukromé sekci třídy deklarujeme hodnotovou proměnnou `bDisposed` typu `bool`, kterou v těle instančního konstruktoru inicializujeme logickou hodnotou `false`. Tato proměnná bude představovat indikátor, jehož pomocí budeme zjišťovat, zda byla, nebo nebyla explicitně aktivována metoda `Dispose` rozhraní `IDisposable`. Naše třída pracuje se dvěma typy zdrojů: řízené zdroje reprezentuje instance třídy `Form` z jmenného prostoru `System::Windows::Forms`. Abychom situaci zjednodušili, jako neřízený zdroj vystupuje proměnná typu `HWND` (což je `typedef` pro `void*`), do které budeme ukládat

manipulátor okna pracovní plochy (tento získáme za asistence Win32 API funkce `GetDesktopWindow`).

POZNÁMKA



Funkci `GetDesktopWindow` v našem případě aktivujeme pomocí softwarové technologie P/Invoke s využitím atributové třídy `DllImportAttribute` z jmenného prostoru `System::Runtime::InteropServices`. Všimněte si, že řízený prototyp nativní funkce je deklarovaný jako statický. Návratovou hodnotou funkce `GetDesktopWindow` je celočíselný identifikátor oka pracovní plochy – ten uchováváme v proměnné `Manipulator` typu `HWND`.

Možná vás zaráží skutečnost, proč se v těle řízené třídy `A` vyskytují dvě verze metody `Dispose`. Nuže, je to proto, že překrytá virtuální metoda by neměla být příliš viditelná – použití přístupového modifikátoru `protected` omezuje viditelnost a použitelnost metody na danou třídu a třídy z ní odvozené, což je rozumné řešení. Naproti tomu, veřejně přístupná bezparametrická metoda `Dispose` je na požádání k dispozici programátorům při realizaci deterministické dealokace objektových zdrojů. Veřejná metoda `Dispose` volá virtuální metodu `Dispose`, která pracuje s jedním formálním parametrem (`Disposing` typu `bool`). Bude-li virtuální metodě `Dispose` předán argument `true`, znamená to, že to byl vývojář, kdo inicioval explicitní proces uvolnění systémových zdrojů. Za těchto okolností dochází k likvidaci řízených (`f->Close();`) i nativních (`Manipulator=0;`) prostředků a k zobrazení informační textové zprávy. Bude-li ovšem virtuální metodě `Dispose` odevzdán argument s hodnotou `false`, znamená to, že ke slovu se dostává implicitní dealokace, kterou nastartovalo běhové prostředí CLR. Za těchto podmínek budou odstraněny pouze nativní zdroje (větev `else` rozhodovacího příkazu `if-else`). Na posledním řádku virtuální metody `Dispose` nastavujeme hodnotu proměnné `bDisposed` na `true`, čímž naznačujeme, že metoda `Dispose` již byla aktivována.

Řízený destruktor třídy `A` (`~A()`) je definovaný docela zajímavě: Místo toho, abychom duplikovali kód virtuální metody `Dispose`, voláme tuto metodu opět, ovšem tentokrát se vstupním argumentem `false`. Tím pádem dochází k dealokaci nativních zdrojů.

K programovému kódu `__gc` třídy `A` bychom si dovolili uvést ještě několik poznámek:

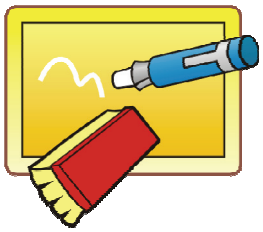
- Zaprvé, když programátor zavolá veřejnou metodu `Dispose`, bude vzápětí aktivována její virtuální kolegyně s argumentem `true`. Aby nedošlo k finalizaci objektu, nebo přesněji, aby automatický správce paměti neaktivoval finalizační metodu, potlačujeme jeho činnost voláním statické metody `SuppressFinalize` tříd `GC`.

- Zadruhé, objekt by po absolvování své finalizace neměl být k dispozici „na použití". Z tohoto důvodu analyzujeme v těle metody `Ziskat_Manipulator` hodnotu proměnné `bDisposed` – jestli byla metoda `Dispose` aktivována, došlo k uvolnění prostředků, a proto není žádoucí objekt znova použít (přestože objekt ještě stále žije na řízené hromadě). Pokud se programátor pokusil k objektu přistupovat i po likvidaci alokovaných zdrojů, musíme přikročit k vygenerování programové výjimky `System::ObjectDisposedException`, které jako argument předáme textovou interpretaci aktivní instance řízené třídy (`this->ToString()`).

Praktická ukázka použití `__gc` třídy `A` by mohla vypadat třeba takhle:

```
// Vytvoření instance __gc třídy A.
GC_Tridy::A __gc * p_A = __gc new GC_Tridy::A();
// Použití objektu pro získání hodnoty manipulátoru.
MessageBox::Show(String::Concat(S"Manipulátor pracovní plochy: ",
p_A->Ziskat_Manipulator()));
// Explicitní dealokace nativních i řízených zdrojů objektu.
p_A->Dispose();
```

POZNÁMKA



Programovací jazyk C++ s Managed Extensions nedisponuje obdobou příkazu `using`, jenž je známý z jazyka C#. Příkaz `using` definuje blok, na konci kterého bude automaticky aktivována metoda `Dispose` vytvořené instance řízené třídy. Jestliže programátor sestrojí pomocí příkazu `using` novou instanci třídy, může si být jistý, že na konci bloku ohraničeném tímto příkazem bude běhovým prostředím CLR zavolána metoda `Dispose` dotyčné instance. Nutnou podmínkou však je, aby použitá třída implementovala rozhraní `IDisposable` a zaváděla definici metody `Dispose`. Více informací vám přináší níže uvedený fragment zdrojového kódu jazyka C#:

```
// Začátek bloku using - Vytvoření objektu třídy
// Graphics.
using (Graphics g = this.CreateGraphics())
{
   // Zde je možné s objektem třídy Graphics pracovat.
} /* Konec bloku using - zde dochází k volání metody
Dispose objektu třídy Graphics.*/
```

Automatická správa paměti – Jeden z pilířů technologie Microsoft .NET Framework

Když společnost Microsoft začátkem 21. století začala programátory, vývojáře a IT odborníky na celém světě zasvěcovat do tajů nové technologické platformy .NET, jedním z průkopníků marketingové kampaně se stala zmínka o automatické správě paměti. Tato novinka se měla stát pomyslnou zbraní těžkého kalibru, kterou se Microsoft snažil radikálně zvýšit pohodlí programátorů migrujících do nově uváděného vývojářského prostředí. Hned na úvod je zapotřebí prohlásit, že automatická správa paměti (angl. Garbage Collection) je skutečným přínosem pro tvůrce softwaru. Každý ostřílený vývojář pracující v nativním C++ totiž ví, kolik problémů mohou napáchat nesprávné anebo nevhodně načasované akce, jež jsou spojené s paměťovým managementem. V minulosti museli být programátoři velice ostražití, neboť na ně poměrně často číhaly mnohé pasti a léčky. Kontrola řádné dealokace obsazených paměťových prostředků však nebyla tou největší překážkou. Daleko tvrdším oříškem byla snaha co možná nejflexibilněji reagovat na různá stádia životních cyklů objektů, což byla činnost, kterou bylo možné nezřídka přirovnat k putování nepřehlednou softwarovou džunglí. Jestli vývojář byť jen na moment přimhouřil oči, bylo velmi pravděpodobné, že zanedlouho se stane kořistí nějaké domorodé potvory. A tak jsme byli konfrontováni s mnoha programovými chybami, předčasnou dealokací zdrojů počínaje a násobnými pokusy o uvolnění obsazených paměťových prostředků konče. Je jistě nepopiratelné, že tyto chyby byly více než nepříjemné, ovšem to pravé zoufalství čekalo na vývojáře teprve ve chvíli, kdy se rozhodl pracovat se soustavou vnořených objektů, a tak se chtě nechtě dostal do křížku s cyklickými referencemi.

Automatická správa paměti, kterou uvádí platforma Microsoft .NET Framework, se snaží najít účinné medikamenty na vzpomínané softwarové choroby. Automatická správa paměti, kterou iniciuje paměťový správce, je pomocným mechanizmem, schopným vyrovnat se s problematickými oblastmi, jež sužovaly vývojáře v nativním prostředí. Začněme tím, že programátoři se již nemusejí zabývat explicitní dealokací řízených zdrojů, jakými jsou například objekty `__gc` tříd. Tato zpráva je bezesporu potěšující, nakolik zbavuje vývojáře povinnosti přímého uvolnění alokovaných objektů a s nimi asociovaných zdrojů. Dalším plusem automatické správy paměti je skutečnost, že se dovede vypořádat s cyklickými referencemi. Řečeno jinými slovy, správa paměti vám dokáže podat pomocnou ruku i v případě, kdy jsou mezi vašimi programovými objekty definovány vzájemné vztahy reprezentované objektovými referencemi. Navzdory tomu, že vybudování mechanizmu na řízení operací spojených s managementem operační paměti má mnoho pozitiv, rozhodně není všelékem, a to zejména v jazyce C++.

Automatická správa paměti je totiž schopna vykonávat svou činnost pouze na řízené hromadě, což je rezervovaná paměťová oblast pro uskladnění řízených prostředků, jakými jsou instance odkazových datových typů, případně také instance hodnotových datových typů uložené v objektových skříňkách. Pokud využíváte ve svých aplikacích nativní zdroje (manipulátory, databázová spojení, mutexy či instance neřízených programových entit), budete si muset s jejich dealokací pomoci sami. Jelikož automatická správa paměti se odehrává pouze na řízené hromadě, není v její kompetenci udělat pořádek také s nativními prostředky. Tato slabá stránka nového paměťového paradigmatu je naneštěstí nejcitelnější právě v jazyce C++ s Managed Extensions. Je to proto, že řízené C++ je jediným .NET-kompatibilním programovacím jazykem, jenž dovoluje paralelní použití řízených i nativních programových instrukcí. Na následujících řádcích si detailně popíšeme algoritmus práce automatické správy paměti na řízené hromadě.

Generační model řízené hromady

Při požadavku na spuštění aplikace .NET je v operační paměti vytvořen fyzický proces, který bude vymezovat exekuční prostor pro nově zaváděnou aplikaci, a který bude rovněž izolovat zpracovávaný programový kód dotyčné řízené aplikace od kódu jiných, ne nutně řízených aplikací. Abychom byli úplně přesní, musíme si povědět, že fyzický proces, jenž je vytvořen v RAM, není jedinou izolační jednotkou aplikací vyhovujícím standardům platformy .NET. Řízené aplikace upotřebují vyspělejší techniku, která pracuje s aplikačními doménami. Aplikační doménu si můžete představit jako logický proces, jenž leží v područí procesu fyzického. Přitom platí pravidlo, že jeden fyzický proces může obsahovat jeden nebo i několik procesů logických. Do aplikační domény je vzápětí umístěno programové vlákno aplikace, které působí jako primární exekuční jednotka. Podle implicitního modelu jsou všechny standardní aplikace pro systém Windows vytvářeny jako jednovláknové moduly se základní úrovní priority. Do programového vlákna je následně napumpován přeložený kód jazyka MSIL, jenž je podroben přímé exekuci.

Prostřednictvím společného běhového prostředí CLR, které kontroluje řízený proces, má aplikace k dispozici speciální paměťovou zónu, které se říká řízená hromada. To, že řízená hromada vystupuje jako kontejner instancí `__gc` tříd, není žádná novinka. Ovšem již málokdo ví, jak řízená hromada ve skutečnosti vypadá a na jakých principech pracuje. První důležitou informací, kterou byste měli mít na paměti je, že chování řízené hromady je založeno na tzv. generačním modelu. To znamená, že řízená hromada není monolitním celkem, nýbrž pozůstává ze tří vzájemně propojených částí. Tyto části se označují termínem generace. Řízená hromada sdružuje tři generace s identifikačním číselným označením 0, 1 a 2. Přestože všechny generace slouží na uchování řízených objektů, každá z nich si rozumí s objekty jiného stáří. Hlavním kritériem pro zařazení jistého

objektu do jedné z uvedených generací je doba životnosti objektu. Podle ní bychom mohli jednotlivé generace řízené hromady charakterizovat takto:

1. Nultá generace – obsahuje nejmladší, právě vytvořené objekty,
2. První generace – seskupuje objekty se středně dlouhou dobou životnosti,
3. Druhá generace – obhospodařuje objekty s nejdelším životním cyklem.

V této souvislosti vás může napadnout otázka, proč je vůbec potřeba členit řízenou hromadu na generace, jež uchovávají rozdílně staré objekty. Máte pravdu, na první pohled může popsaný model vzbuzovat pochybnosti, avšak tyto jsou zcela zbytečné. Generační model je propracovaným systémem, který zabezpečuje nejenom potřebnou diferenciaci řízených instancí, ale také velice napomáhá co možná nejefektivnější činnosti automatického správce paměti. Správa paměti totiž vychází z premisy, že kontrola jedné generace je mnohem rychlejší než kontrola všech generací, a tedy celé řízené hromady. Nejčastěji se analýza provádí na nulté generaci, v níž jsou situovány nejmladší objekty. V určitých případech je však nutno zkontrolovat hned několik generací najednou, výjimečně dokonce všechny tři. Kompletní kontrola všech dostupných objektových generací se vyskytuje především při přeplněné řízené hromadě a jako taková je spíše výjimečná.

Algoritmus práce automatického správce paměti je veden parciální analýzou, která dosahuje při práci s objekty `__gc` tříd lepší výkonnostní charakteristiky. Každá z uvedené trojice generací disponuje rozdílným alokačním prostorem. Pod pojmem alokační prostor rozumíme paměťovou oblast (jednoznačně identifikovatelnou prostřednictvím adres), která má určitou kapacitu (měřenou zpravidla v kilobajtech nebo megabajtech). Každá generace tak dovede poskytnout útočiště jenom předem stanovenému počtu objektů, jehož hodnota se odvíjí od jejich kapacitní náročnosti. Prostor řízené hromady je alokovaný okamžitě po startu aplikace .NET, protože již tehdy dochází k tvorbě základních aplikačních objektů.

Životní cyklus objektu se začíná jeho vytvořením pomocí operátoru `__gc new` (přesněji zavoláním instrukce `newobj` jazyka MSIL). Čerstvě založený objekt se nachází na začátku svého doby životnosti, a proto je umístěn do nulté generace řízené hromady. Z uvedeného vyplývá, že generace č. 0 je jakousi vstupní branou do světa řízené hromady, neboť do této generace jsou automaticky ukládány všechny nově zkonstruované objekty. Po přidání objektu do nulté generace dochází ke spuštění instančního konstruktoru (buď implicitního nebo explicitního), jenž objekt inicializuje a uvádí jej do provozuschopného stavu. V další etapě se může objekt soustředit na realizaci těch úkonů, pro které byl naprogramován.

I když jsme si pověděli, že všechny nové objekty jsou ukládány do nulté generace, vyvstává otázka, jak ve skutečnosti řízená hromada ví, kde má nově alokovaný objekt umístit. Pro tento účel používá řízená hromada speciální ukazatel `NextObjPtr`, jenž označuje paměťovou adresu, na kterou lze nový objekt uložit. Jakmile je objekt úspěšně alokován, hodnota ukazatele `NextObjPtr` se dynamicky změní a ukazatel je od této chvíle nasměrován na novou adresu, na kterou je možné uložit další nový objekt.

Algoritmus práce automatického správce paměti

Automatický správce paměti sleduje jednotlivé generace řízené hromady, přičemž na základě tzv. kořenů sestavuje referenční stromy pro všechny přítomné objekty. Referenční strom je něco jako stromová struktura kořenů, kterou si můžeme zjednodušeně popsat jako soustavu objektových referencí uložených v odkazových proměnných. Prostřednictvím referenčních stromů dovede správce paměti vždy jednoznačně určit, zda je kýžený objekt využívaný kódem řízené aplikace či nikoliv. Pokud existuje alespoň jedna reference směrující na objekt, říkáme, že takovýto objekt je

dosažitelný z programového kódu. Dosažitelné objekty považuje automatický správce paměti za živé, a proto není možné, aby je z řízené hromady odstranil. Na druhou stranu existují také objekty nedosažitelné, což jsou objekty, na které již nejsou navázány žádné aplikační kořeny. Nedosažitelné objekty jsou předmětem uvolňovacího procesu, v rámci kterého jich správce paměti identifikuje a uskuteční jejich likvidaci.

Na základě dosud představených poznatků bychom mohli nabýt dojmu, že objekty jsou v podstatě buď dosažitelné, nebo nedosažitelné, přičemž osud těch druhých je v plné moci správce paměti. Bohužel, situace je ve skutečnosti o něco složitější.

Zdrojem komplikací se stává potřeba finalizace objektu, tedy nutnost exekuce finalizační metody instance daného datového typu. Pokaždé, když dojde k vytvoření objektu, jenž definuje svou finalizační metodu, automatický správce paměti přidá odkaz na tento objekt do tzv. finalizačního seznamu. Finalizační seznam je jednoduchá datová struktura, která obsahuje odkazy na všechny objekty vyžadující explicitní finalizaci. Pomocí finalizačního seznamu správce paměti ví, že specifikované objekty bude nutné podrobit finalizačnímu procesu, a tedy že je nelze jednoduše z řízené hromady uvolnit v okamžiku, kdy se stanou nedosažitelnými.

Pokusme se nyní blíže podívat na práci automatického správce paměti. Jak jsme již naznačili, každá generace řízené hromady má k dispozici jistý alokační prostor, do něhož lze ukládat řízené objekty. Stejně tak víte, že všechny objekty jsou po svém vytvoření uloženy do nulté generace. Společné běhové prostředí CLR pokračuje v alokaci objektů v nulté generaci, dokud není vyčerpán všechen dostupný přidělený alokační prostor. Za těchto okolností správce paměti iniciuje uskuteční analýzy generace č. 0 s cílem zjistit počet nedosažitelných objektů. Je možné, že v procesu skenování nulté generace správce paměti identifikuje několik nedosažitelných objektů. Tyto již aplikační kód nepoužívá, a proto by bylo možné je uvolnit. Nedosažitelné objekty, které nevyžadují explicitní finalizaci, mohou být dealokovány bez jakýchkoliv potíží. Ovšem u nedosažitelných objektů s nutností explicitní finalizace uplatňuje správce paměti jiný přístup. V rámci první kolekce, tedy první analýzy nulté generace, jsou takovéto objekty rozpoznány a reference na ně jsou z finalizačního seznamu přesunuty do dalšího seznamu, jenž seskupuje objekty připravené na finalizaci. Nedosažitelné objekty tak nejsou v rámci prvního průběhu automatické správy paměti uvolněny. Je to konec konců logické: správce paměti nemůže objekty uvolnit, protože je nevyhnutné aktivovat jejich finalizační metody. Správce paměti finalizační metody všech objektů skutečně zavolá a následně odstraní reference na tyto objekty ze seznamu objektů připravených na finalizaci. Při další analýze nulté generace řízené hromady správce paměti zjistí, že objekty už prošly svou finalizací, což znamená, že v této chvíli jsou již opravdovým odpadem. Jelikož neexistuje žádný smysluplný důvod pro to, aby byly nepotřebné objekty i nadále uskladněny na řízené hromadě, automatický správce paměti je jednoduše zlikviduje.

Při interakci s objekty vyžadujícími finalizaci pracuje automatický správce paměti ve dvou krocích, co je příčinou potenciální výkonnostní penalizace. Proto zavádějte finalizační metody pouze v opravdu odůvodněných případech.

Procesy kolekce

Algoritmus práce automatického správce paměti v spojitosti s nultou generací řízené hromady jsme si vysvětlili v předcházející kapitole. Co se však stane, když se alokační kapacita generace č. 0 vyčerpá? Tuto okolnost zaregistruje společné běhové prostředí CLR, které aktivuje automatického správce paměti. Správce paměti vykoná analýzu dostupných objektů a zjistí, které z nich jsou nedosažitelné z programového kódu. Nedosažitelné objekty, které nevyžadují explicitní finalizaci, jsou zlikvidovány a ty objekty, které si vynucují volání svých finalizačních metod jsou zase zpracovány na samostatném programovém vláknu. Řízené objekty, které přežijí kolekci nulté generace

řízené hromady, jsou přemístěny do generace s vyšším pořadovým číslem, tedy do generace č. 1. Poté, co je transport vybraných řízených objektů z nulté generace úspěšně proveden, je nutno uskutečnit následující dvě akce:

1. **Defragmentace operační paměti.** Vedlejším produktem přesunu objektů s delší dobou životnosti z nulté generace do generace první je vznik paměťových děr, čili jakýchsi prázdných míst, které byly dosud alokovány řízenými objekty. Efekt vzniku paměťových děr je samozřejmě nepříznivý, protože narušuje doteď celistvý charakter řízené hromady. Je proto přirozené, že se automatický správce paměti snaží tento neduh eliminovat. V závislosti na počtu a pozici transportovaných řízených objektů může být řízená hromada různě fragmentována. Správce paměti proto přikročí k její defragmentaci, což znamená, že přeskupí objekty tak, aby následovaly v řade za sebou. Po zdárné defragmentaci je paměť řízené hromady opět souvislá a tvoří jeden kompaktní blok objektů.

2. **Aktualizace objektových referencí.** Každý objekt, který se nacházel v nulté generaci řízené hromady byl kdykoliv dosažitelný prostřednictvím typově silné objektové reference, která byla uložena v příslušné odkazové proměnné jistého datového typu. Pokud bychom se podívali na objektovou referenci pod drobnohledem, zjistili bychom, že jde ve skutečnosti o speciální ukazatel determinující paměťovou adresu, na níž se objekt nachází. Nicméně, po procesu kolekce jsou stále živé objekty přesunuty do generace č. 1, což znamená, že „staré" objektové reference směrující do nulté generace, již nejsou dále aktuální, a tedy je nelze použít. Této skutečnosti si je vědom také správce paměti, a proto jakmile je ukončen přesun objektů z nulté do první generace, dochází k aktualizaci objektových referencí. Tak je zaručeno, že všechny přesunuté objekty budou i nadále dosažitelné z programového kódu, přestože již změnily své působiště.

Po první kolekci je nultá generace řízené hromady prázdná, a je tedy volná pro další objekty, jejichž zrození bude aplikace .NET požadovat. Když se však generace č. 0 opět zaplní, bude muset být zase povolaný automatický správce paměti. Ten iniciuje realizaci další kolekce, přičemž zjišťuje poměr aktivních a nepotřebných objektů. Další postup již probíhá podle známého scénáře: objekty, které přežijí fázi skenování a analýzy jsou přesunuty do generace č. 1, zatímco nedosažitelné objekty jsou uvolněny (anebo připraveny na uvolnění). Dobrá, ovšem co se děje v generaci č. 1? Podobně jako nultá generace, také generace č. 1 disponuje předem definovaným alokačním prostorem, jenž stanovuje její kapacitní náročnost. Generace č. 1 je obvykle schopna uchovat větší počet objektů než nultá generace, protože může využívat větší porci operační paměti. Před výše uvedenou druhou kolekcí nulté generace se automatický správce paměti podívá na obsah generace č. 1. Jelikož můžeme předpokládat, že v této objektové generaci se nenacházejí objekty s alokační kapacitou přesahující mezní hranici, správce paměti se nebude zatěžovat provedením kolekce generace č. 1, ale místo toho svou pozornost soustředí pouze na nultou generaci. Popsaný postup je garancí vyšší pracovní produktivity správce paměti, protože uskutečnění kolekce jedné generace je vždy rychlejší, než kdyby mělo dojít ke dvěma kolekcím v generacích č. 0 a 1.

Po vykonání druhé kolekce jsou dosažitelné objekty z nulté generace převedeny do generace č. 1. Za předpokladu, že v generaci č. 1 bude dostatek paměťového prostoru pro uložení dalších a dalších objektů, správce paměti nebude tuto objektovou generaci zatěžovat procesy kolekce. A to dokonce ani tehdy ne, když se budou v této generaci nacházet již nepotřebné objekty, ke kterým už programový kód nepřistupuje.

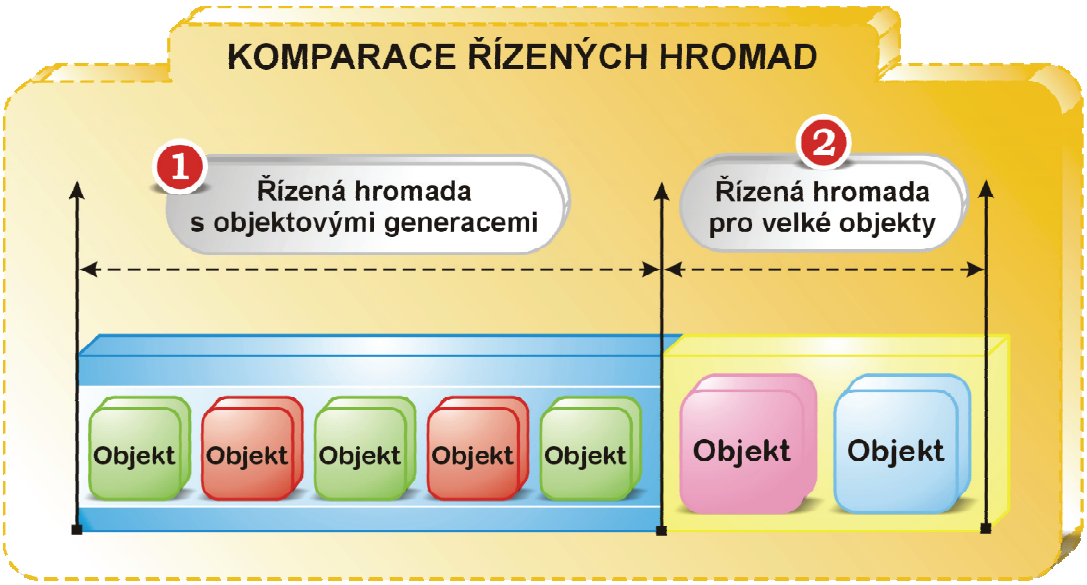
Samozřejmě, ani alokační kapacita generace č. 1 není nekonečná, a proto dříve nebo později dojde k jejímu celkovému zaplnění. Pokud bude doručen požadavek na alokaci nového objektu, přičemž obě generace (0 a 1) budou plně vytížené, automatický správce paměti nařídí provedení kolekci v generaci č. 1 a také v generaci č. 0. Všechny dosažitelné řízené objekty, které odolají procesu kolekce, budou přemístěny do generace

č. 2. Analogicky, nepotřebné objekty ležící v generaci č. 1 budou označeny jako odpad a nastane jejich uvolnění (ať už implicitní nebo explicitní).

Generace č. 2 reprezentuje objektovou generaci s nejvyšším pořadovým číslem, což znamená, že žádné další generace již neexistují. V druhé generaci najdou své útočiště dlouho žijící objekty, jejichž životní cykly se mohou překrývat s dobou životnosti celé aplikace .NET. Generace č. 2 je svým způsobem výjimečná: disponuje totiž největším alokačním prostorem ze všech objektových generací. Objekty, které se jednou dostanou do generace č. 2, v ní zůstanou až do okamžiku, než budou automatickým správcem paměti odstraněny. Po změně pozice objektů na řízené hromadě je provedena defragmentace paměti společně s aktualizací objektových referencí.

Pokud bychom měli jednotlivé generace řízené hromady porovnat podle frekvence výskytu kolekcí, mohli bychom prohlásit, že nejčastěji skenovanou generací je ta s nulovým identifikátorem. Naopak, generace č. 2 je vzhledem ke své velikosti podrobována procesům kolekce s mnohem menší frekvencí.

Automatický správce paměti alokuje speciální řízenou hromadu pro uskladnění velikých objektů. Mějte prosím na paměti, že tato hromada není součástí řízené hromady se třemi objektovými generacemi. Ve skutečnosti jde o nezávislou virtuální paměť. Termínem veliký objekt se označuje objekt, který obsazuje relativně velký alokační prostor: správce paměti považuje za veliký každý objekt s kapacitou větší než 20 KB. Ačkoliv jsou veliké objekty pod dohledem správce paměti, nejsou přesouvány prostřednictvím objektových generací – to je základní rozdíl, jenž bychom identifikovali, kdybychom porovnali architektonickou stavbu řízené hromady s objektovými generacemi a speciální řízené hromady pro velké objekty. Druhým odlišným znakem je skutečnost, že speciální řízená hromada pro velké objekty není nikdy seskupována, anebo jinak řečeno defragmentována. Je to způsobeno tím, že realizace přeskupování velkých objektů by vyústila do vyšší zátěže procesoru počítačové stanice, což není žádoucí. Je-li jednou velký objekt na řízenou hromadu umístěn, zůstává v ní až dokud není prostřednictvím automatického správce paměti dealokován.



Obr. 1.5: Ilustrace řízených hromad, které kontroluje správce paměti

Získání informací o automatické správě paměti pomocí systémového nástroje Performance Monitor

Pokud patříte mezi zkušené programátory v jazyce C++ s Managed Extensions, jistě jste již pomýšleli na to, jak získat o procesu automatické správy paměti více informací. Skvělým začátkem je použití aplikace **Performance Monitor Command Line Shell** (anebo zkráceně pouze **Performance Monitor**), ke které se dostanete následovně:

1. Otevřete **Ovládací panely** a poklepejte na ikonu **Nástroje pro správu**.

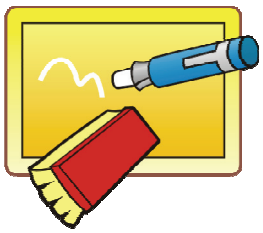
2. Aplikaci **Performance Monitor** spustíte aktivací zástupce **Výkon**.

TIP	
	Program **Performance Monitor** můžete spustit také přímo z Průzkumníka: jde o spustitelný soubor perfmon.exe, který je umístěn ve složce Windows\System32.

Aplikaci **Performance Monitor** použijeme společně s testovací aplikací .NET s názvem Cpp_13_01. Naše aplikace není vůbec složitá: tvoří ji formulář, na němž se nachází jedno tlačítko (instance ovládacího prvku `Button`). V těle zpracovatele události `Click` tlačítka je vložen následující fragment programového kódu:

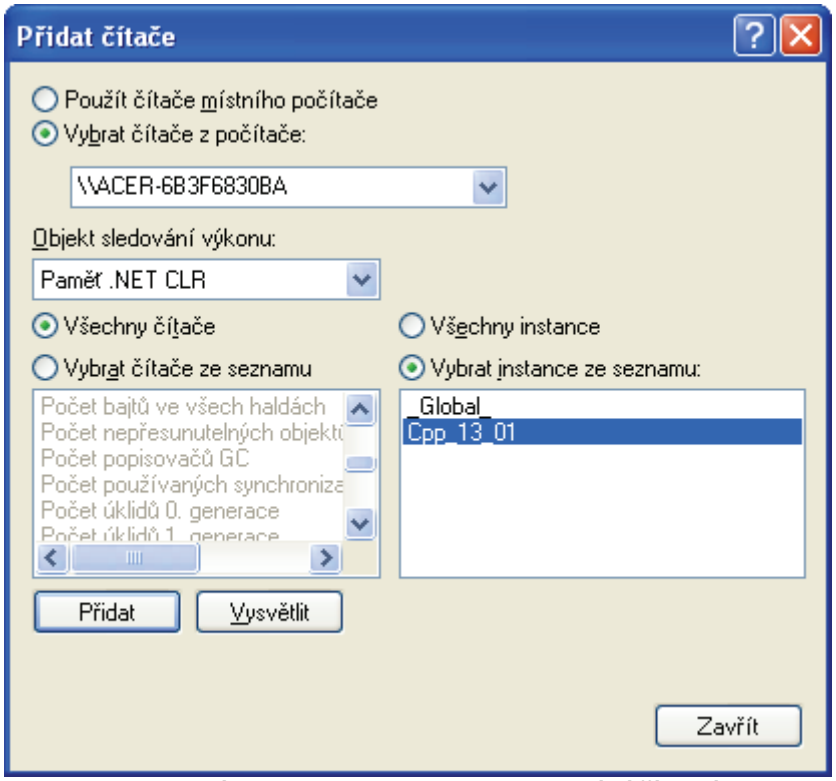
```
static Byte n = 1;
for(Int32 i=1; i < 100000; ++i)
Object __gc * obj = __gc new Object();
MessageBox::Show(String::Concat(
S"Celkový počet vytvořených objektů: ", (n * i).ToString(),
S"."), S"Test řízené hromady", MessageBoxButtons::OK,
MessageBoxIcon::Information);
++n;
```

Tento kód budeme využívat při monitorování činnosti automatické správy paměti. Kdykoliv dojde ke stisknutí tlačítka na formuláři, bude sestrojených 100-tisíc řízených objektů třídy `System::Object`. Tyto objekty budou uloženy do nulté generace řízené hromady. Na zásobníku bude alokován prostor pro stejný počet odkazových proměnných typu `System::Object __gc*`. Vztah mezi odkazovými proměnnými a řízenými objekty lze charakterizovat korelací 1:1, což znamená, že každá odkazová proměnná obsahuje typově silnou objektovou referenci nasměrovanou na právě jeden řízený objekt. Proces založení instancí provází zobrazení informační zprávy, ve které se uživatel dozví, kolik objektů bylo ve skutečnosti vytvořených. Zrod instancí je inkrementální: po každé aktivaci tlačítka bude sestrojených dalších sto tisíc řízených objektů a odkazových proměnných, přičemž informační dialog nás seznamuje s celkovým počtem zrozených objektů.

POZNÁMKA	
	Aplikace **Performance Monitor** nám dovoluje sledovat využití automatické správy paměti libovolné aplikace .NET. Budete-li tedy chtít, můžete použít svou aplikaci a pozorovat využití řízení hromady a proces automatické správy paměti na ní.

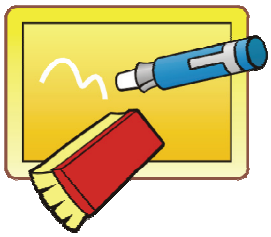
Pokračujte dle těchto instrukcí:

1. V pravé části okna programu **Performance Monitor** vyhledejte tlačítko **Přidat** (na tlačítku je zobrazený symbol plus (+)). Stejně tak můžete vyvolat klávesovou zkratku CTRL+I.
2. Na obrazovce počítače se objeví dialogové okno **Přidat čítače**.
3. Ze seznamu **Objekt sledování výkonu** vyberte možnost **Paměť .NET CLR**.
4. Pod seznamem **Objekt sledování výkonu** vyberte položku **Všechny čítače**.
5. Ujistěte se, že je zvolena položka **Vybrat instance ze seznamu**, která se nachází v pravé části dialogového okna. Pod touto položkou je umístěn seznam, jenž obsahuje dvě položky: **_Global_** a název spuštěné aplikace .NET.
6. Ze seznamu vyberte název řízené aplikace, čímž nařídíte, aby cílem monitorovacích služeb byla právě ona.
7. Pokud jste provedli všechny výše uvedené operace, dialogové okno by mělo vypadat takto:



Obr. 6: Nastavení konfigurace pro monitorování řízené hromady

POZNÁMKA



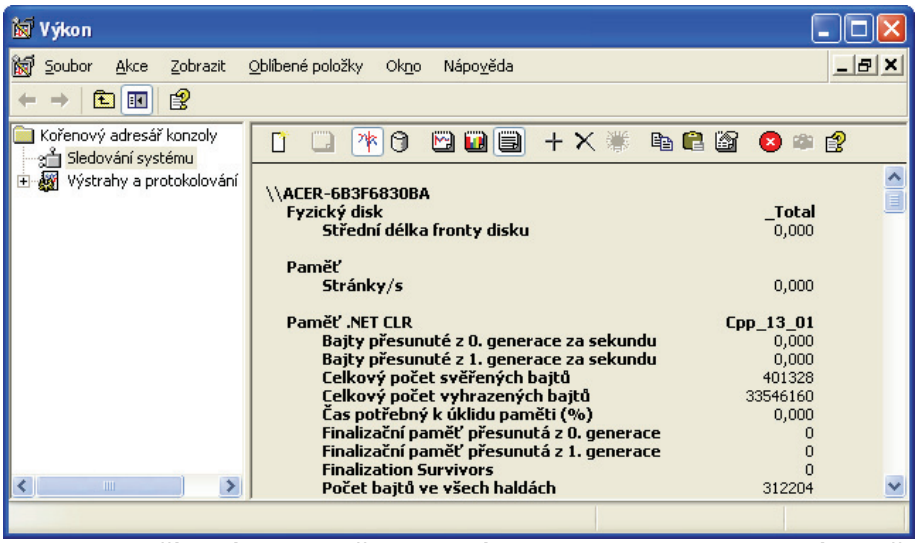
Je-li aktivní volba **Vybrat čítače ze seznamu**, můžete přistupovat ke všem čítačům asociovaným s činností automatické správy paměti a vybírat z nich ty, o kterých si přejete obdržet více informací. Tak například čítač **Počet vyvolání GC** zobrazuje nejvyšší počet kolekcí, jež byly uskutečněny v důsledku explicitního volání metody **Collect** třídy **GC** z jmenného prostoru **System**.

TIP

Seznam **Vybrat čítače ze seznamu** obsahuje poměrně velké množství někdy až záhadně znějících položek. Pokud nebudete vědět, co ta-která položka ve skutečnosti znamená, vyberte ji a následně klepněte na tlačítko **Vysvětlit**. Téměř okamžitě se zviditelní dialogové okno **Vysvětlit text** s nápovědou charakterizující zvolený čítač.

8. Klepněte na tlačítko **Přidat**. Jakmile tak učiníte, do seznamu sledovaných čítačů aplikace **Performance Monitor** budou umístěny všechny interní čítače, které se zaměřují na analýzu využití řízené hromady aplikace .NET. Dialogové okno se však automaticky nezavře, a proto jej musíte aktivací tlačítka **Zavřít** odstranit sami.

Čítače jsou standardně znázorněné v docela nepřehledném seznamu (nacházejícím se ve spodní části okna aplikace), přičemž společně s vývojem sledovaných parametrů se zobrazuje také jejich grafická interpretace. Lepší rozvržení předmětných informací přináší režim **Zobrazit zprávu**, který můžete aktivovat klepnutím na tlačítko s notýskem ( ). Jste-li fanoušky klávesových zkratek, do režimu zpráv můžete vstoupit rovněž pomocí „dvojhmatu" CTRL+R. V této chvíli by mělo okno aplikace **Performance Monitor** mít tuto podobu:



Obr. 7: Informace o řízené hromadě pokusné aplikace .NET zobrazené v režimu zpráv

Monitorovací služby pracují podobně jako rentgen, protože vám dovolují nahlížet i na ty nejmenší detaily, jež identifikují pracovní algoritmus automatické správy paměti. Jak se můžete sami přesvědčit, správa paměti je aktivována bezprostředně po spuštění

testovací aplikace. Klepněte na tlačítko na aplikačním formuláři a sledujte, jak se mění hodnoty jednotlivých čítačů. Budete-li chtít vytvořit dalších sto tisíc objektů, opět stiskněte tlačítko. Hodnoty čítačů budou odrážet skutečnou vytíženost automatické správy paměti.

Část 2 - Programovací jazyk C++/CLI

C++/CLI – zrození nového řízeného C++

Přestože programovací jazyk C++ s Managed Extensions je plnohodnotným prostředkem pro psaní aplikací .NET pomocí řízených rozšíření jazyka C++, jeho celková syntaktická struktura a koncepční rozvržení nejsou úplně ideální. Vývojáři jsou inteligentní osoby a tuto skutečnost tudíž zaregistrovali velice rychle. Z vlastních zkušeností víme, že tábor tvůrců zdrojového kódu se při diskusi na téma „C++ a .NET" zvykl rozdělit na několik fragmentů. Jedna skupina vývojářů si stěžovala na překomplikovanou syntaktickou podobu programovacího jazyka, která dle jejich mínění ještě více stěžovala pochopení již tak docela nepřehledného zdrojového kódu. Pro další partii programátorů byla trnem v oku nekonzistentnost jazyka, která vystupovala do popředí zejména při míchání nativního a řízeného kódu. Kdesi v rohu místnosti si svůj tábor postavili příznivci jazyka C#, kteří prohlašovali, že řízené C++ je příliš těžkopádné na to, aby si mohlo měřit síly s jejich oblíbeným mřížkovým céčkem. Nuže, kdepak je tedy pravda?

Ačkoliv C++ s Managed Extensions je životaschopný programovací jazyk pro vytváření řízených aplikací .NET, je nutno přiznat, že výtky programátorů byly v mnoha směrech opodstatněné. Ba co víc, rovněž samotní tvůrci jazyka C++ s Managed Extensions cítili, že řízenému C++ je zapotřebí vdechnout nový život. A jak řekli, tak také udělali a výsledkem jejich práce je zcela přepracovaný programovací jazyk s názvem C++/CLI, jenž se může pyšnit plnou konformitou se společnou jazykovou infrastrukturou (Common Language Infrastructure, CLI) vývojově- exekuční platformy Microsoft .NET Framework 2.0.

Jazyk C++/CLI je součástí softwarového produktu Visual C++ 2005, který patří do rodiny vývojářských nástrojů nového Visual Studia (rovněž s přídomkem 2005). Pokud je pro vás komplexní Visual Studio 2005 příliš robustní, možná uvítáte sdělení, že programovací jazyk C++/CLI můžete používat také ve Visual C++ 2005 Express. Tento produkt je „odlehčenou" verzí mocného Visual C++ 2005, přičemž je určen především pro studenty informačních technologií, hobby vývojáře a programátory-začátečníky. Podobně jako jeho větší bráška, také Visual C++ 2005 Express disponuje integrovaným vývojovým prostředím a samozřejmě plnou podporou syntaxe jazyka C++/CLI. Navíc si jej můžete opatřit zdarma pouhým stažením z webových stránek společnosti Microsoft. Tak získáte výtečný nástroj, s nímž můžete odstartovat svou kariéru vývojáře v řízeném C++!

Jazyk C++/CLI představuje rozšíření nativního programovacího jazyka C++, jenž byl standardizován organizací ISO (jak se praví v dokumentu ISO/IEC 14882:2003 – Programming languages – C++). Tvůrci jazyka C++/CLI se soustředili hlavně na naplnění následujících cílů:

- zabezpečit elegantní syntaxi a sémantiku, která bude přirozená a snadno pochopitelná pro programátory v C++,
- začlenit prvotřídní podporu pro programovací rysy společné jazykové infrastruktury (CLI), k nimž patří kupříkladu práce s hodnotovými a odkazovými datovými typy, generické programování a automatická správa paměti,
- přenést vše dobré z nativního C++: tím se míní možnost transportu standardních C++ programových rysů, které si u komunity vývojářů získaly značnou oblibu,
- eliminovat výskyt složitých a krkolomných syntaktických a sémantických elementů známých z C++ s Managed Extensions.

Jazyk C++/CLI je vskutku velikým krokem kupředu, což jak pevně věříme zjistíte ihned poté, co se sžijete s novinkami přijatého jazykového standardu. V této části naší příručky

si představíme základní novinky jazyka C++/CLI, s nimiž se musí obeznámit všichni vývojáři, kteří aktuálně používají C++ s Managed Extensions. Budeme se věnovat důležitým oblastem, jako jsou datové typy a objektově orientované programování. Předem bychom vás ovšem chtěli poprosit, abyste přehled novinek nepokládali za vyčerpávající kurz nového řízeného C++. Kdybychom se vydali na tuto cestu, naše vývojářská brožura by se rázem proměnila na tlustou knihu, kterou by bylo nutné opatřit tvrdou vazbou. Naší ambicí je provést vás nejzajímavějšími inovacemi jazyka C++/CLI v „programátorsky" přívětivé formě, přičemž doufáme, že ve vás vzbudíme zájem o další studium!

Začínáme s C++/CLI

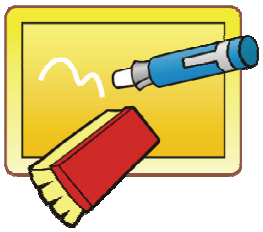
Domníváme se, že nejlepším začátkem bude vytvoření první jednoduché aplikace .NET v programovacím jazyce C++/CLI. Spusťte tedy Visual Studio 2005 nebo Visual C++ 2005 Express a pomocí projektové šablony **CLR Console Application** založte nový projekt pro konzolovou aplikaci. Tvorba projektu je velice snadná, neboť průvodce pro vás vygeneruje všechny nezbytné součásti softwarového řešení, které uspořádá do přehledné stromové struktury. Jakmile je průvodce hotov se všemi úkony, zobrazí zdrojový kód implementačního souboru (s extenzí .cpp) jazyka C++/CLI v editoru zdrojového kódu. Automaticky vytvořený kód vypadá následovně:

```
#include "stdafx.h"

using namespace System;

int main(array<System::String ^> ^args)
{
    Console::WriteLine(L"Hello World");
    return 0;
}
```

Jelikož jsme si přáli založit projekt konzolové aplikace, veškeré textové výstupy našeho kódu budou směrovány do konzolového okna. Vstupním bodem našeho programu je funkce `main`, což není překvapující. Novinkou je však zápis formálního parametru funkce `main`, který deklaruje referenci na řízené pole řízených ukazatelů schopných uchovávat objektové reference na instance třídy `System::String`. Ano, víme, že na první pohled se může deklarace uvedeného formálního parametru jevit přímo hrůzostrašně, ovšem nepropadejte panice. O změnách v řízených ukazatelích a polích si zanedlouho povíme víc. Nyní naši pozornost přenesme na řádek, v němž dochází k volání statické metody `WriteLine` třídy `Console`. Této metodě je předána řetězcová konstanta, před níž stojí prefix `L`.

UPOZORNĚNÍ	
	Prefix `L` definuje „široké" řetězcové literály. Nicméně to, že se objevuje na tomto místě, je pravděpodobně pochybení průvodce při autogenerování výchozího kódu projektu konzolové aplikace. Jak si možné vzpomínáte, v jazyce C++ s Managed Extensions jsme deklarovali řízené řetězcové literály pomocí prefixu `S`. Řízené textové řetězce, jež byly vybaveny prefixem `S`, byly mnohem efektivněji spravovány než nativní protějšky s příznakem `L` (při použití prefixu `S` nebyl kompilátorem jazyka C++ s Managed Extensions generován dodatečný režijní programový kód). Pokud budete chtít v jazyce C++/CLI deklarovat řízenou řetězcovou konstantu, nemusíte prefix `S` uvádět (a prefix `L` už teprve ne).

POZNÁMKA



Když konzolovou aplikaci spustíte, v okně se objeví specifikovaný textový řetězec. Bohužel, pravděpodobně jej sotva postřehnete, neboť okno konzole se ihned po vypsání řetězce automaticky uzavře. Tento problém můžete vyřešit vložením příkazu, jenž volá statickou metodu `Read` třídy `Console`:

```
#include "stdafx.h"

using namespace System;

int main(array<System::String ^> ^args)
{
    Console::WriteLine("Hello World");
    // Zde čekáme na načtení dalšího znaku...
    Console::Read();
    return 0;
}
```

Po zavolání metody `Console::Read` se program pozastaví, přičemž bude čekat na zadání dalšího textového znaku. Stiskněte tedy libovolnou klávesu a svoji volbu potvrďte klávesou ENTER, na což okno aplikace zmizí z obrazovky.

Zkonstruovaná konzolová aplikace je velice jednoduchá, ovšem v tuto chvíli je pro nás důležitější skutečnost, že tato aplikace plně využívá všech výhod jazyka C++/CLI jakožto i bázové knihovny třídy platformy .NET Framework 2.0. Překladač jazyka C++/CLI a sestavovací program (linker) pracují společně, aby byl výsledkem jejich práce spustitelný soubor, jenž je naplněn metadaty a instrukcemi jazyka MSIL. Řečeno jinak, na výstupu získáváme řízený modul, který vznikl díky aktivovanému přepínači `/clr`. Pokud zalovíte v paměti, jistě si vzpomenete, že podobný přepínač jsme měli také v jazyce C++ s Managed Extensions. Ano, to je sice pravda, no měli byste vědět, že kompilátor jazyka C++/CLI vám nabízí širší možnosti v oblasti obsahové konfigurace sestavených řízených programů. Vedle přepínače `/clr` totiž uvádí další přepínače, jejichž pomocí lze precizněji upravit chování a obsahovou strukturu vygenerovaných modulů. Přehled dostupných přepínačů kompilátoru jazyka C++/CLI můžete vidět v tab. 2.1.

Tab. 2.1: Přehled přepínačů kompilátoru jazyka C++/CLI

Přepínač	Textový popis přepínače	Charakteristika
/clr	Common Language Runtime Support	Tento přepínač aktivuje podporu běhového prostředí CLR a společné jazykové infrastruktury platformy .NET Framework 2.0. Je vybrán implicitně po zvolení .NET-kompatibilní projektové šablony. Zapíná podporu pro jazyk C++/CLI. Řízené moduly sestavené při aktivaci přepínače `/clr` obsahují metadata, MSIL kód a porci nativního kódu pro načtení a spuštění běhového prostředí CLR. Kromě toho mohou obsahovat také nativní programový kód, díky čemuž se vygenerovaná sestavení často nazývají „smíšenými" nebo „mixovanými". Vzájemná spolupráce mezi nativním a řízeným kódem C++ je možná za přispění technologie C++ Interop. Přepínač `/clr` je dobrou volbou zejména v případech, kdy máte nativní C++ aplikaci a přejete si ji velice rychle přenést do řízeného prostředí.
/clr:pure	Pure MSIL Common Language Runtime Support	Přepínač `/clr:pure` je striktnější než přepínač `/clr`. Je to proto, že sestavené řízené moduly nesmí obsahovat jakékoliv nativní funkce (ačkoliv mohou zavádět definice nativních datových typů). Na druhé straně mohou samozřejmě obsahovat řízené datové typy a procedury. Přepínač `/clr:pure` znemožňuje použití technologie C++ Interop, no programátoři mohou kooperovat s nativními knihovnami DLL přes softwarovou vrstvu P/Invoke.
/clr:safe	Safe MSIL Common Language Runtime Support	Finálním produktem přepínače `/clr:safe` jsou řízená sestavení, která je možné podrobit verifikačními procesu (podobně jako je tomu u sestavení, jež jsou připraveny kompilátory jazyků Visual Basic a C#). Verifikovatelná sestavení jsou ponímána jako bezpečná, což znamená, že jejich životní cyklus je ovlivněn bezpečnostní politikou a nastaveními v prostředí operačního systému. Zhotovené řízené moduly nesmějí zahrnovat žádné nativní datové typy nebo neřízené funkce a rovněž tak nelze použít ani technologii C++ Interop.
/clr:oldSyntax	Common Language Runtime Support, Old Syntax	Přepínač `/clr:oldSyntax` je jako časový portál: když do něj vstoupíte, přenese vás do dob, kdy vládl jazyk C++ s Managed Extensions. Řečeno méně metaforicky, když zapnete tento přepínač, můžete i nadále psát kód jazyka C++ s Managed Extensions, jenž v dané relaci nahrazuje jazyk C++/CLI.

Pomineme-li použití posledního přepínače `/clr:oldSyntax`, tak máme před sebou tři přepínače, které nám dovolují pečlivě upravovat chování naší řízené aplikace. Přitom platí, že přepínač `/clr:pure` je přísnější než `/clr` a přepínač `/clr:safe` je zase přísnější než `/clr:pure`. Pokud byste rádi produkovali plně kvalifikované řízené moduly, jež budou vyhovovat bezpečnostním kriteriím stanovených konfigurací společného běhového

prostředí CLR, můžete zvolit přepínač `/clr:safe`. V tomto případě se však předpokládá, že budete upotřebovat pouze řízený kód jazyka C++/CLI. Naopak, je-li vaším cílem sestrojení aplikace .NET, která bude kromě řízeného kódu obsahovat i nepřehlédnutelné množství nativních zdrojových instrukcí, patrně sáhnete po přepínači `/clr`. Někde uprostřed si svou pozici hájí přepínač `/clr:pure`, jenž je sice primárně zaměřen na řízený kód, ovšem připouští také volání nativních funkcí. Docela zajímavá je komparace dovedností různých typů aplikací, které byly zkompilovány pomocí rozličných přepínačů kompilátora jazyka C++/CLI. Více informací přináší tab. 2.2.

Tab. 2.2: Jaké jsou možnosti řízených aplikací při použití přepínačů /clr, /clr:pure a /clr:safe

Dovednost	Přepínač kompilátoru jazyka C++/CLI		
	/clr	/clr:pure	/clr:safe
Použití bázové knihovny tříd	✓	✓	✓
Použití knihovny CRT	✓	✓	✗
Použití knihovny MFC/ATL	✓	✗	✗
Definice neřízených funkcí	✓	✗	✗
Definice neřízených datových typů	✓	✓	✗
Možnost volání z nativního kódu	✓	✗	✗
Možnost volání nativních funkcí	✓	Pouze funkce ve stylu jazyka C	Pouze technologie P/Invoke
Podpora reflexe	Pouze knihovny DLL	✓	✓

Přehled syntaktických inovací jazyka C++/CLI

Programovací jazyk C++/CLI je moderním programovacím nástrojem a jako takový přinesl nepřeberné množství hlubokých syntakticko-sémantických inovací, změn a vylepšení. V této části se pokusíme poukázat na nejdůležitější z nich.

Hodnotové a odkazové datové typy v C++/CLI

Jelikož je jazyk C++/CLI podobně jako C++ s Managed Extensions postaven na vývojově-exekuční platformě Microsoft .NET Framework, umožňuje vývojářům pracovat s hodnotovými a odkazovými (referenčními) datovými typy. Abychom si výklad poněkud zjednodušili, rozdělíme obě zmíněné kategorie typů na dvě skupiny: základní (primitivní) typy a uživatelsky definované typy.

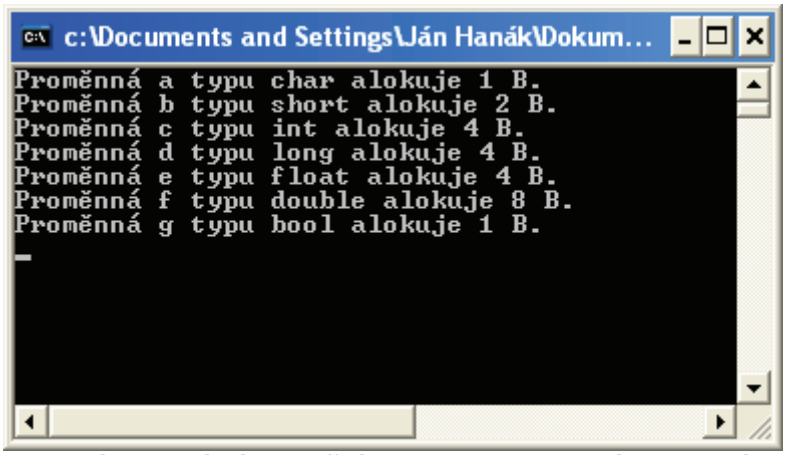
K primitivním hodnotovým datovým typům řadíme typy, které jsou tak běžné, že je kompilátor jazyka C++/CLI bez potíží rozezná. Jedná se o typy pro práci s textovými znaky (`char`), celočíselnými hodnotami (`short`, `int`, `long`) či reálnými čísly (`float`, `double`). Jestliže použijete v programovém kódu zástupce těchto datových typů, kompilátor jazyka C++/CLI je zcela automaticky převedete na jejich řízené protějšky, jimiž jsou `System::SByte`, `System::Int16`, `System::Int32`, `System::Int32` s volbou `modopt IsLong`, `System::Single` a `System::Double`. Podobně můžete pracovat s datovým typem `bool` pro reprezentaci logických hodnot `true` a `false`. Systémovým

ekvivalentem typu `bool` je `System::Boolean`. Základní hodnotové datové typy jsou substituovány odpovídajícími řízenými typy, ovšem je nutno připomenout, že když použijete rezervované klíčové slovo jazyka C++, jako třeba `int`, konkrétní substituce je závislá na implementaci. To znamená, že pouze implementace říká, zda bude typ `int` nahrazen typem `System::Int32` nebo `System::Int64`. Na druhou stranu, když explicitně použijete systémový datový typ, třeba `System::Int32`, deklarovaná instance bude na jakékoliv implementaci vyžadovat alokaci dvaatřiceti bitů.

Nominální alokační kapacitu primitivních datových typů můžeme zjistit pomocí operátoru `sizeof`. Níže uvedený výpis zdrojového kódu ukazuje, jak lze tento operátor použít v souvislosti s několika deklarovanými proměnnými.

```
#include "stdafx.h"

using namespace System;

int main(array<System::String ^> ^args)
{
   char a = 'A'; short b = 10; int c = 20; long d = 30;
   float e = 11.56f; double f = 22.222;
   bool g = false;
   Console::WriteLine(String::Concat("Proměnná a typu char alokuje ",
      sizeof(a), " B."));
   Console::WriteLine(String::Concat("Proměnná b typu short alokuje ",
      sizeof(short), " B."));
   Console::WriteLine(String::Concat("Proměnná c typu int alokuje ",
      sizeof(int), " B."));
   Console::WriteLine(String::Concat("Proměnná d typu long alokuje ",
      sizeof(long), " B."));
   Console::WriteLine(String::Concat("Proměnná e typu float alokuje ",
      sizeof(float), " B."));
   Console::WriteLine(String::Concat("Proměnná f typu double alokuje ",
      sizeof(double), " B."));
   Console::WriteLine(String::Concat("Proměnná g typu bool alokuje ",
      sizeof(bool), " B."));
   Console::Read();
   return 0;
}
```

Výstup kódu můžete vidět na obr. 2.1.



Obr. 2.1: Zobrazení nominální alokační kapacity primitivních datových typů jazyka C++/CLI

Rezervovaná klíčová slova jazyka C++/CLI pro určení datových typů se ve skutečnosti konvertují na příslušné hodnotové třídy uložené v bázové knihovně tříd (tyto hodnotové třídy vystupují v pozici systémových ekvivalentů primitivních typů C++/CLI). V tab. 2.3 můžete pozorovat vzájemný vztah mezi základními typy jazyka C++/CLI a jejich systémovými protějšky.

Tab. 2.3: Vazba mezi primitivními typy jazyka C++/CLI a odpovídajícími hodnotovými třídami bázové knihovny tříd vývojově-exekuční platformy .NET Framework 2.0

Datový typ jazyka C++/CLI	Systémový ekvivalent	Popis
`bool`	`System::Boolean`	Logická hodnota pravda (`true`) nebo nepravda (`false`)
`char`	`System::SByte` (s volbou `modopt IsSignUnspecifiedByte`)	8bitová celočíselní hodnota
`signed char`	`System::SByte`	8bitová celočíselní se znaménkem
`unsigned char`	`System::Byte`	8bitová celočíselní hodnota bez znaménka
`short`	`System::Int16`	16bitová celočíselní hodnota se znaménkem
`unsigned short`	`System::UInt16`	16bitová celočíselní hodnota bez znaménka
`int`	`System::Int32`	32bitová celočíselní hodnota se znaménkem
`unsigned int`	`System::UInt32`	32bitová celočíselní hodnota bez znaménka
`long`	`System::Int32` s volbou `modopt IsLong`	32bitová celočíselní se znaménkem
`unsigned long`	`System::UInt32` s volbou `modopt IsLong`	32bitová celočíselní hodnota bez znaménka
`long long int`	`System::Int64`	64bitová celočíselní hodnota se znaménkem
`unsigned long long int`	`System::UInt64`	64bitová celočíselní hodnota bez znaménka
`float`	`System::Single`	Hodnota v pohyblivé řádové čárce s jednoduchou přesností
`double`	`System::Double`	Hodnota v pohyblivé řádové čárce s dvojitou přesností
`long double`	`System::Double` s volbou `modopt IsLong`	Hodnota v pohyblivé řádové čárce s mimořádně vysokou přesností
`wchar_t`	`System::Char`	Textový znak znakové sady Unicode

Mluvíme-li o hodnotových datových typech, musíme vzpomenout ještě typ `System::Decimal`, jenž je vhodný především pro náročné finanční kalkulace. Typ `System::Decimal` však nepatří mezi primitivní datové typy. Na rozdíl od jazyka C#, proměnnou typu `System::Decimal` nemůžeme v jazyce C++/CLI explicitně inicializovat pomocí sufixu `M`. Místo toho můžeme upotřebit tento postup:

```
System::Decimal dec;
dec = (Decimal)10.11;
MessageBox::Show("Hodnota proměnné dec je " + dec + ".");
```

V okně se zprávou bude zobrazeno desetinné číslo 10,11, což je nejspíš to, co bychom očekávali. Když se ale lépe podíváte na podobu textového argumentu, který předáváme metodě `Show` třídy `MessageBox`, jistě vám neunikne nový syntaktický zápis zřetězování

dat typu `System::String`. Ano, vážení přátelé, nyní můžeme také v jazyce C++/CLI spájet textové řetězce pomocí přetíženého aritmetického operátoru `+`. Je to stejně pohodlné jako v C# a navíc mnohem efektivnější.

Odkazový datový typ System::String

Práce s textovými řetězci byla od minula citelně vylepšena, z čehož budete míst jistě velikou radost. Začněme konstatováním, že před řetězcové konstanty již nemusíte přidávat předponu `S` pro to, abyste z nich vykouzlili řízené textové řetězce. Kompilátor jazyka C++/CLI je nyní dost chytrý na to, aby si uvědomil, kdy hodláte použít řízený textový řetězec. Pouze připomeňme, že všechny textové řetězce jsou na platformě .NET Framework 2.0 reprezentovány instancemi neboli objekty vestavěné třídy `System::String` a jako takové žijí na řízené hromadě. To znamená, že když se rozhodneme vytvořit textový řetězec, musíme mít připravenou rovněž adekvátní odkazovou proměnnou, do níž uložíme odpovídající objektovou referenci. Představme si nyní jednoduchou ukázku sestrojení uživatelsky definovaného textového řetězce:

```
String^ retezec = "Programovací jazyk C++/CLI";
```

Hned na první pohled můžete pozorovat zásadní změny týkající se deklarace odkazové proměnné. Typem této proměnné již není `String __gc*`, jak tomu bylo v jazyce C++ s Managed Extensions, nýbrž `String^`. Co to znamená? Tvůrci jazyka C++/CLI usoudili, že je nutno zřetelně odlišit řízené ukazatele na instance řízených tříd (které sídlí na řízené hromadě) od klasických nativních ukazatelů, které mohou směrovat takřka kamkoliv. Proto dochází k uvedení nového symbolu `^`, jenž reprezentuje tzv. sledovací manipulátor (angl. tracking handle). Sledovací manipulátor si můžete představit jako nástupce řízeného ukazatele `__gc*` z jazyka C++ s Managed Extensions. Manipulátor `T^` může být nasměrován pouze na instanci řízeného typu `T`, jehož instance jsou alokovány na řízené hromadě běhového prostředí CLR. Pro migrující vývojáře tedy platí snadno pochopitelné pravidlo: řízený `__gc*` ukazatel je nyní nutno nahradit symbolem `^`, jehož pomocí je deklarován sledovací manipulátor. Ačkoliv více si o tvorbě řízených objektů budeme povídat za chvíli, bude dobré, když si tuto novinku uvědomíte již nyní.

Odkazová proměnná `String^` uchovává sledovací manipulátor, jenž směruje na instanci třídy `System::String`, v níž je uložen textový řetězec. Podstatou nově uváděného manipulátoru je schopnost kdykoliv „vystopovat" předmětnou instanci na řízené hromadě. Jak jistě víte, objekty na řízené hromadě se mohou přesouvat mezi jednotlivými generacemi v závislosti na aktuálním stadiu svého životního cyklu. Manipulátor `^` je syntaktickým symbolem, který upozorňuje na skutečnost, že instance umístěné na řízené hromadě sice mohou měnit své pozice, no přesto budou kdykoliv spolehlivě dosažitelné.

Novinkou je rovněž zřetězování většího počtu řetězcových literálů pomocí přetíženého aritmetického operátoru pro sčítání (`+`). Programátoři již proto nejsou nuceni explicitně volat metodu `Concat` třídy `String` a předávat jí řetězcové argumenty.

```
String^ str1 = "Programovací ";
String^ str2 = "jazyk ";
String^ str3 = "C++/CLI.";
// Zřetězení textových řetězců pomocí operátoru +.
String^ str4 = str1 + str2 + str3;
Console::WriteLine(str4);
```

Díky jednodušší a přehlednější syntaxi je seskupování textových řetězců mnohem pohodlnější, než tomu bylo v jazyce C++ s Managed Extensions.

Určitě se shodneme na tom, že programový kód, jenž pro spájení textových řetězců volá přetížený operátor `+`, je velice líbivý. Nicméně bude dobré, když budete vědět, jak věci doopravdy fungují. Pokud přetransformujeme výše znázorněný kód jazyka C++/CLI do ryzích MSIL instrukcí, dojdeme k pozoruhodnému poznání:

```
.method assembly static int32  main(string[] args) cil managed
{
  // Code size       54 (0x36)
  .maxstack  2
  .locals ([0] string str4,
           [1] string str3,
           [2] string str2,
           [3] string str1)
  IL_0000:  ldnull
  IL_0001:  stloc.3
  IL_0002:  ldnull
  IL_0003:  stloc.2
  IL_0004:  ldnull
  IL_0005:  stloc.1
  IL_0006:  ldnull
  IL_0007:  stloc.0
  IL_0008:  ldstr   bytearray (50 00 72 00 6F 00 67 00 72 00 61 00 6D 00 6F
  00   // P.r.o.g.r.a.m.o.
                                 76 00 61 00 63 00 ED 00 20 00 )
  // v.a.c... .
  IL_000d:  stloc.3
  IL_000e:  ldstr      "jazyk "
  IL_0013:  stloc.2
  IL_0014:  ldstr      "C++/CLI."
  IL_0019:  stloc.1
  IL_001a:  ldloc.3
  IL_001b:  ldloc.2
  IL_001c:  call       string [mscorlib]System.String::Concat(string,
                                                              string)
  IL_0021:  ldloc.1
  IL_0022:  call       string [mscorlib]System.String::Concat(string,
                                                              string)
  IL_0027:  stloc.0
  IL_0028:  ldloc.0
  IL_0029:  call       void [mscorlib]System.Console::WriteLine(string)
  IL_002e:  call       int32 [mscorlib]System.Console::Read()
  IL_0033:  pop
  IL_0034:  ldc.i4.0
  IL_0035:  ret
} // end of method 'Global Functions'::main
```

V kódu jazyka MSIL můžeme hezky pozorovat veškerý tok kódu. Nejprve dochází k alokaci čtyř odkazových proměnných typu `string`. Tyto proměnné jsou pojmenovány jako `str1`, `str2`, `str3` a `str4`, přičemž každé proměnné je přiřazena celočíselní identifikace podle tohoto vzoru: `str1=3`, `str2=2`, `str3=1` a `str4=0`. V dalších krocích jsou na zásobník uloženy nulové reference (instrukce `ldnull`), jimiž budou deklarované proměnné inicializované. Příkaz `stloc.[i]` (kde `i` je číselní identifikátor dotyčné proměnné) provádí vyjmutí hodnoty ze zásobníku a její uložení do lokální proměnné. Jelikož na zásobník jsou nejprve umístěny nulové reference, jsou to právě ony, které putují do lokálních proměnných. Jakmile jsou proměnné typu `string` inicializovány, je vykonána instrukce `ldstr bytearray`, která nařizuje načíst řetězcovou konstantu uloženou v bajtovém poli. Na zásobníku se v této chvíli nachází první textový řetězec, jenž je exekucí instrukce `stloc` uložen do odkazové proměnné typu `string`. Pak je načten druhý řetězec, a také on je vložen do příslušné proměnné. Tatáž posloupnost se

opakuje rovněž pro třetí, a tudíž poslední textový řetězec. Tím pádem jsou ve všech lokálních proměnných uloženy jednotlivé kousky textu. Nyní na scénu vstupuje instrukce `ldloc` – ta je odpovědná za bezproblémové načtení lokální proměnné na zásobník. Jak si můžete všimnout, jako první je načtena proměnná `str1` (instrukce `ldloc.3`) a za ní přichází proměnná `str2` (instrukce `ldloc.2`). Když se na zásobníku nacházejí tyto proměnné, běhové prostředí CLR volá metodu `Concat` třídy `System.String`, které předává odkazy na dva textové řetězce.

Metoda `Concat` provede zřetězení a nový řetězec, jenž vznikl spojením dvou řetězců (umístěných v lokálních proměnných `str1` a `str2`), je vrácen v podobě návratové hodnoty metody. Poté je instrukcí `ldloc.1` načtena na zásobník proměnná `str3` a zase je zavolána metoda `Concat`, která připojí textový řetězec k stávajícímu řetězci, jenž byl stvořen při předchozím volání této metody. Nyní již máme k dispozici finální řetězec, jenž je tvořen třemi dílčími textovými řetězci. Finální řetězec uposlechne instrukci `stloc.0`, která zajišťuje jeho uložení do lokální proměnné `str4` typu `string`.

Studiem zbývajících instrukcí se již nemusíme obtěžovat, protože hlavní pointa zdrojového kódu byla objasněna. Jak jsme se díky tomuto laboratornímu cvičení přesvědčili, pod povrchem je i nadále volána metoda `String::Concat`, a to bez toho, aby o tom programátor pracující v jazyce C++/CLI vůbec věděl. To je však v pořádku, protože většinu vývojářů nemusí zajímat nízkoúrovňová interpretace zřetězování textových literálů – přednější je skutečnost, že C++/CLI umožňuje flexibilnější provádění běžných operací s textovými řetězci.

Odkazový datový typ System::Object a mechanizmus sjednocení typů

Třída `System::Object` je primární bázovou třídou, která je rodičem všech vestavěných i nově definovaných programových tříd. Instance třídy `System::Object` mohou uchovávat data hodnotových a odkazových datových typů. V prvním případě, kdy má být do instance třídy `System::Object` uložena hodnota hodnotového datového typu (hodnotové struktury, hodnotové třídy či primitivního typu jako je `short` nebo `int`), musí být nastartován mechanizmus sjednocení typů. Úkolem mechanizmu sjednocení typů je přimět instanci hodnotového typu, aby se chovala jako objekt. Sjednocení typů vyžaduje dynamické sestrojení řízené instance typu `System::Object` na řízené hromadě. Založená instance je označována jako objektová skříňka. Když je objektová skříňka na světě, je běhovým prostředím CLR nastartován kopírovací proces, během kterého dochází k duplikaci původní hodnoty hodnotové instance a jejímu umístění do objektové skříňky. Nakonec je vrácen řízený ukazatel na objektovou skříňku, jenž je uložen do připravené odkazové proměnné. Takto jsme charakterizovali mechanizmus sjednocení typů v první části této vývojářské příručky, kdy jsme jej zkoumali za asistence jazyka C++ s Managed Extensions. Jaká je ovšem situace v jazyce C++/CLI?

Samozřejmě, také zde se s mechanizmem sjednocení typů setkáte. První změnou je konstatování, že objektová skříňka již nebude dosažitelná přes řízený ukazatel typu `Object __gc*`, ale pomocí sledovacího manipulátoru `Object^`. Druhou modifikací je, že v prostředí jazyka C++/CLI je mechanizmus sjednocení typů realizován implicitně, což znamená, že jej nemusíte explicitně startovat pomocí příkazu `__box`, jak jste to dělali v C++ s Managed Extensions. Abychom vám představili nové prvky, představíme si provedení mechanizmu sjednocení typů v obou zmíněných programovacích jazycích.

Ukázka 1: Aktivace mechanizmu sjednocení typů v jazyce C++ s Managed Extensions

```
Int32 i = 100;
Object __gc* obj = __box(i);
MessageBox::Show(String::Concat(S"Hodnota proměnné obj je ",
   obj, S"."), S"Mechanizmus sjednocení typů", MessageBoxButtons::OK,
   MessageBoxIcon::Information);
```

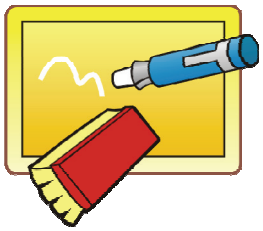
Když jsme chtěli vložit celočíselnou hodnotu proměnné typu `Int32` do odkazové proměnné typu `Object __gc*`, museli jsme použít příkaz `__box`, jenž inicioval spuštění mechanizmu sjednocení typů. Kdybychom tento příkaz v kódu neuvedli, kompilátor by nás zastavil s chybovým hlášením. Explicitní zápis příkazu `__box` v jazyce C++ s Managed Extensions poukazoval na nemožnost provedení implicitní konverze hodnot zúčastněných datových typů.

Ukázka 2: Aktivace mechanizmu sjednocení typů v jazyce C++/CLI

```
Int32 i = 100;
Object^ obj = i;
MessageBox::Show("Hodnota proměnné obj je " + obj + ".",
   "Mechanizmus sjednocení typů", MessageBoxButtons::OK,
   MessageBoxIcon::Information);
```

V jazyce C++/CLI je situace jiná, přičemž hned si všimnete, že funkčně ekvivalentní zdrojový kód se může pyšnit mnohem přehlednějším a srozumitelnějším zápisem. Věnujme se však druhému řádku kódu, v němž je nastartován mechanizmus sjednocení typů. Jak můžete vidět, tento mechanizmus je spuštěn zcela implicitně, aniž bychom byli nuceni použít příkaz `__box`. Ve skutečnosti je použití příkazu `__box` zakázáno, takže kdybyste jej i náhodou zapsali, kompilátor by vás na chybu upozornil.

POZNÁMKA



Pro úplnost dodejme, že klíčové slovo `__box` můžete v kódu jazyka C++/CLI aplikovat pouze tehdy, je-li aktivován přepínač kompilátoru `/clr:oldSyntax`. V tomto případě vám kompilátor Visual C++ 2005 dovolí psát starý kód jazyka C++ s Managed Extensions, no tuto možnost vám nedoporučujeme využívat. Konec konců, proč byste se chtěli vracet zpátky, když v jazyce C++/CLI na vás čekají nové a vzrušující programové prvky?

Dobrá, mechanizmus sjednocení typů je po novém prováděn automaticky, a to kdykoliv kompilátor jazyka C++/CLI usoudí, že je nutno jej aktivovat. Tím získávají programátoři v C++/CLI stejný komfort jako jejich kolegové využívající jazyky C# nebo Visual Basic 2005.

Interní realizace mechanizmu sjednocení typů je v jazyce C++/CLI shodná s výkonem tohoto mechanizmu v C++ s Managed Extensions. To znamená, že na řízené hromadě bude sestrojena objektová skříňka a celočíselní hodnota typu `Int32` bude nakopírována do této skříňky. Změnou ovšem je, že po provedení mechanizmu sjednocení typů nebude vrácen řízený ukazatel typu `Object __gc*`, ale sledovací manipulátor typu `Object^`. Tato modifikace je odvozena od implementace nových syntaktických identifikátorů a rozšíření, ovšem nijak neovlivňuje skutečný chod mechanizmu sjednocení typů.

Zpětný chod mechanizmu sjednocení typů

Výsledným produktem práce mechanizmu sjednocení typů je alokace objektové skříňky na řízené hromadě a umístění kopie původní hodnoty hodnotového typu do této skříňky. Pokud nastane situace, kdy bude zapotřebí vybrat hodnotu z objektové skříňky, ke slovu se dostane zpětný chod mechanizmu sjednocení typů. Podívejme se nejprve, jak jsme získávali hodnotu z objektové skříňky v jazyce C++ s Managed Extensions.

Zpětný chod mechanizmu sjednocení typů v C++ s Managed Extensions

```
Int32 i = 100;
// Nejdříve hodnotu typu Int32 zabalíme a vložíme do objektové skříňky...
Object __gc* obj = __box(i);
// ...a pak ji rozbalíme a uložíme do další hodnotové proměnné.
Int32 j = *dynamic_cast<__box Int32 __gc*>(obj);
Console::WriteLine(j.ToString());
Console::Read();
```

Je pochopitelné, že abychom mohli hodnotu z objektové skříňky vyjmout, musí nějaká objektová skříňka vůbec existovat a rovněž tak musí být naplněna specifikovanou hodnotou hodnotového datového typu. Jakmile jsou tyto podmínky splněny, můžeme přistoupit k získání hodnoty z objektové skříňky. Jak víte, jazyk C++ s Managed Extensions nenabízí žádnou přímočarou cestu k provedení této operace. Proto musíme uskutečnit konverzi pomocí operátoru `dynamic_cast<>`, díky které získáme řízený ukazatel na zabalenou hodnotu typu `Int32` (jde o ukazatel `__box Int32 __gc*`). Máme-li po ruce zmíněný řízený ukazatel, můžeme jej prostřednictvím operátoru `*` dereferencovat, na což dostáváme kýženou celočíselnou hodnotu (100 v našem případě). Tu pak zobrazujeme v příkazovém řádku konzolové aplikace.

Zpětný chod mechanizmu sjednocení typů v C++/CLI

Jazyk C++/CLI s sebou nese sofistikované inovace, s nimiž vám uskutečňování zpětného chodu mechanizmu sjednocení typů přijde jako příjemná zábava. Kdybychom chtěli výše uvedenou ukázku jazyka C++ s Managed Extensions portovat do nového prostředí, počínali bychom si asi takto:

```
Int32 i = 100;
// Zde je implicitně spuštěn mechanizmus sjednocení typů...
Object^ obj = i;
// ...a zde je zase získána kopie hodnoty uskladněné v objektové skříňce.
Int32 j = *dynamic_cast<Int32^>(obj);
Console::WriteLine("Hodnota proměnné j je " + j);
Console::Read();
```

Ačkoliv jsme opět vsadili na operátor `dynamic_cast<>`, můžete postřehnout, že syntaxe je mnohem přehlednější a přirozenější. Jednoduše se ptáme, zda objektová skříňka uchovává zabalenou celočíselnou hodnotu typu `Int32`, a pokud ano, dereferencujeme sledovací manipulátor typu `Int32^`, čímž získáváme přístup ke kýžené hodnotě. Přestože můžeme být se syntaktickou podobou konverzní operace docela spokojeni, měli byste vědět, že v jazyce C++/CLI můžeme jít ještě dál. Máme-li chuť, můžeme zpětný chod mechanizmu sjednocení typů aktivovat, aniž bychom použili operátor `dynamic_cast<>`. Říkáte, že to není možné? Nuže, následující fragment zdrojového kódu se vás pokusí přesvědčit.

```
Int32 i = 100;
Object^ obj = i;
// Zde zahajujeme zpětný chod mechanizmu sjednocení typů.
// Všimněte si, že místo operátoru dynamic_cast<> provádíme jednoduchou
// konverzní operaci ve stylu jazyka C.
Int32 j = (Int32)obj;
Console::WriteLine("Hodnota proměnné j je " + j);
Console::Read();
```

Kdybyste něco takového udělali v kódu jazyka C++ s Managed Extensions, kompilátor by se docela rozzlobil. V C++/CLI jde o regulérní operaci, jejímž důsledkem je generování instrukce `unbox [mscorlib]System.Int32` jazyka MSIL.

UPOZORNĚNÍ

Při realizaci zpětného běhu mechanizmu sjednocení typů pomocí výše nastíněné konverzní operace ve stylu jazyka C dochází k použití operátoru `safe_cast<>`. Přestože není použití tohoto operátoru v kódu zřejmé, kompilátor jazyka C++/CLI nahrazuje konverzi ve stylu C voláním operátoru `safe_cast<>`. Tento operátor mohou vývojáři přímo použít ve zdrojovém kódu a dle oficiálních informací se jedná o preferovanou techniku. Přepsaná verze předcházejícího výpisu zdrojového kódu by pak vypadala takhle:

```
Int32 i = 100;
Object^ obj = i;
Int32 j = safe_cast<Int32>(obj);
Console::WriteLine("Hodnota proměnné j je " + j);
Console::Read();
```

Jelikož je mechanizmus sjednocení typů proveden kdykoliv dojde k uložení hodnoty hodnotového typu do odkazové proměnné, programátoři se musí mít na pozoru v okamžiku, kdy se chystají naznačit, že jistá odkazová proměnná neobsahuje objektovou referenci na instanci určitého odkazového datového typu. Když jste chtěli v jazyce C++ s Managed Extensions deklarovat takovouto odkazovou proměnnou, mohli jste to udělat následovně:

```
// Odkazová proměnná je inicializována nulovou hodnotou, což znamená,
// že není nasměrována na žádnou instanci odkazového datového typu.
Object __gc * obj = 0;
```

Takto jste nařídili, že odkazová proměnná `obj` typu `Object __gc*` byla „vynulována“, což znamená, že byla zlikvidována objektová reference, která v ní mohla být uložena.

Možná se domníváte, že výše zapsaný přiřazovací příkaz bychom mohli v jazyce C++/CLI přepsat takto:

```
// Co znamená tento příkaz v C++/CLI?
Object^ obj = 0;
```

Tento příkaz pro přiřazení je zcela legitimní a kompilátor proti němu neřekne ani muk. Nicméně, jeho skutečný význam je zcela jiný, než jsme zamýšleli. Tento příkaz totiž neodstraňuje objektovou referenci z proměnné `obj`, ale nařizuje implicitní provedení mechanizmu sjednocení typů. To tedy znamená, že na řízené hromadě bude alokována objektová skříňka, do níž bude umístěna duplicitní nulová hodnota. Nakonec bude

navrácen sledovací manipulátor typu `Object^`, jehož pomocí bude sestrojená objektová skříňka dosažitelná.

Když budete chtít inicializovat odkazovou proměnnou typu `Object^` nulovou objektovou referencí, musíte použít nové klíčové slovo jazyka C++/CLI, jímž je `nullptr`:

```
// Nyní je to správně...
Object^ obj = nullptr;
```

Klíčové slovo `nullptr` uloží do odkazové proměnné prázdnou objektovou referenci, která není nasměrována na žádný objekt. Ve skutečnosti je `nullptr` ekvivalentem hodnoty `null` v jazyce C# či klíčového slova `Nothing` ve Visual Basicu 2005.

Enumerační (výčtové) datové typy

Enumerační typy se v jazyce C++ s Managed Extensions definují jako `__value enum`. Jazyk C++/CLI přichází s novou syntaktickou jazykovou specifikací, která odstraňuje použití symbolu dvojitého podtržení (__) před názvy klíčových slov. V C++/CLI se enumerační typy definují jako enumerační třídy pomocí identifikátoru `enum class`.

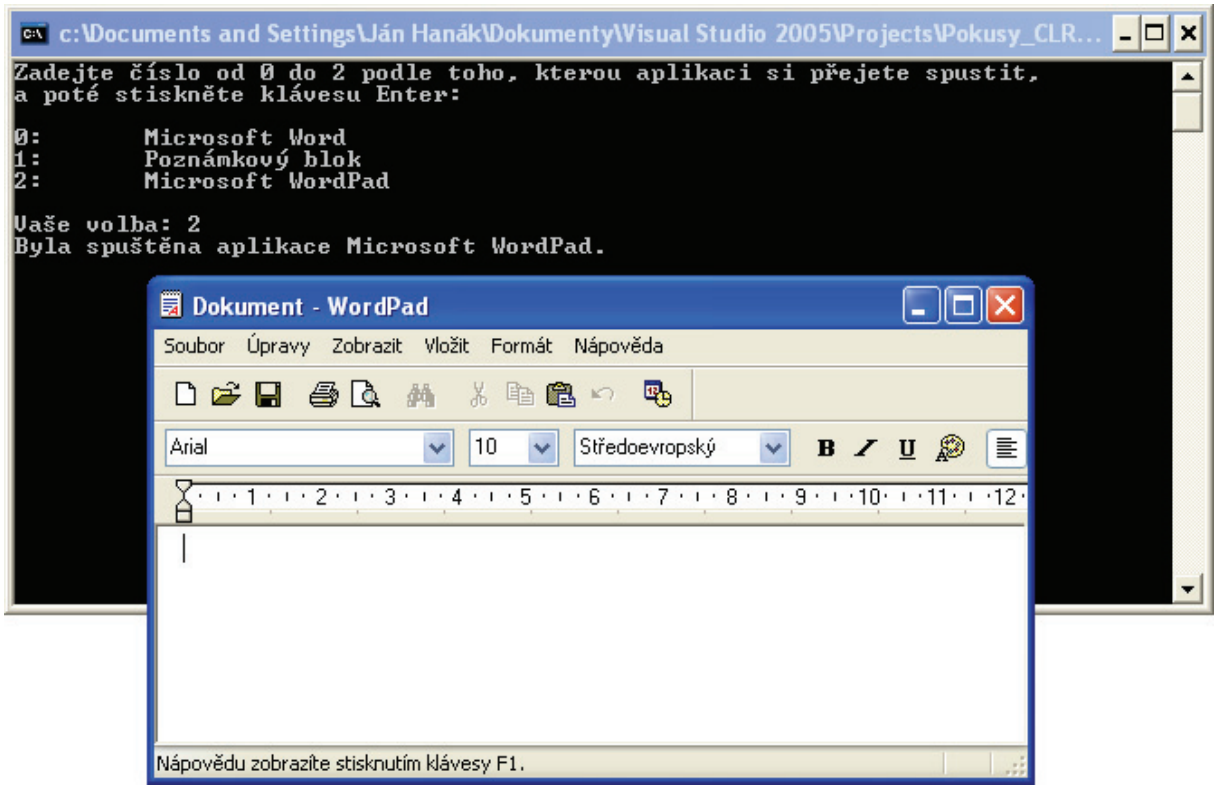
```
enum class SeznamAplikaci
{
      Microsoft_Word, Poznamkovy_blok, Microsoft_WordPad
};
```

Pro členy výčtového typu platí stejná inicializační pravidla jako v nativním C++ či C++ s Managed Extensions. Pokud nejsou jednotlivé enumerátory explicitně inicializovány celočíselnými hodnotami, kompilátor jazyka C++/CLI zabezpečí jejich implicitní inicializaci podle této posloupnosti: prvnímu členu enumerace bude přiřazena hodnota 0, druhému 1, třetímu 2 atd. Enumerační typ můžeme snadno využít v programovém kódu:

```
SeznamAplikaci App;
// Zobrazení uživatelské volby.
Console::WriteLine("Zadejte číslo od 0 do 2 podle toho, kterou aplikaci " +
   "si přejete spustit,\na poté stiskněte klávesu Enter:\n" +
   "\n" + "0:\t Microsoft Word" + "\n" + "1:\t Poznámkový blok" +
   "\n" + "2:\t Microsoft WordPad" + "\n");
Console::Write("Vaše volba: ");
// Uchování číselního identifikátoru zadané volby.
String^ Odpoved = Console::ReadLine();
// Konverze uživatelského vstupu na hodnotu enumeračního typu.
App = (SeznamAplikaci)Convert::ToSByte(Odpoved);
switch(App)
{
   case 0:
       System::Diagnostics::Process::Start("WINWORD");
       Console::WriteLine("Byla spuštěna aplikace Microsoft Word.");
       break;
   case 1:
       System::Diagnostics::Process::Start("Notepad");
       Console::WriteLine("Byla spuštěna aplikace Poznámkový blok.");
       break;
   case 2:
       System::Diagnostics::Process::Start("Wordpad");
       Console::WriteLine("Byla spuštěna aplikace Microsoft WordPad.");
       break;
}
```

```
Console::Read();
```

Naše enumerace deklaruje tři datové členy, které reprezentují číselné identifikace různých textových procesorů. Když se program rozeběhne, vybízíme uživatele, aby zadal číslo, které odpovídá cílové aplikaci, která bude záhy spuštěna. Jakmile uživatel provede svůj výběr, program vytiskne správu o specifikované aplikaci a tuto aplikaci také skutečně spustí (obr. 2.2).



Obr. 2.2: Program jazyka C++/CLI využívající enumeraci `SeznamAplikaci`

Enumerační typy jsou i nadále hodnotovými typy jazyka C++/CLI, přičemž jsou odvozeny od třídy `System::Enum`, která zase dědí své charakteristiky ze třídy `System::ValueType`. Podobně můžeme také v jazyce C++/CLI definovat enumeraci, jejíž enumerátory jsou založeny na určitém bázovém datovém typu.

```
// Definice enumeračního typu s explicitním určením bázového datového
// typu enumerátorů.
public enum class CiselneSoustavy : SByte
{
   BINARNI = 2,
   OKTALOVA = 8,
   DECIMALNI = 10,
   HEXADECIMALNI = 16
};
```

Zajímavé je, že vedle definičního příkazu `enum class` lze upotřebit také příkaz `enum struct`:

```
// Enumeraci lze definovat i pomocí příkazu enum struct.
private enum struct Planety
{
   Mars, Jupiter, Saturn, Pluto, Zeme,
   Venuse, Uran, Neptun, Merkur
};
```

V tomto případě jsme definovali výčtový typ `Planety`, jehož pojmenované konstanty představují vesmírná tělesa v našem universu.

Hodnotové třídy a hodnotové struktury

V jazyce C++ s Managed Extensions se hodnotové třídy definovaly příkazem `__value class`, zatímco hodnotové struktury se vytvářely pomocí příkazu `__value struct`. Jazyk C++/CLI posílá symboly dvojitého podtržení do věčných lovišť, čehož důsledkem jsou nové příkazy `value class` a `value struct`. Ze sémantického hlediska jsou hodnotové třídy a hodnotové struktury takřka identické. Obě entity mohou definovat datové položky uchovávající data a členské metody, které realizují s těmito daty programové operace. Rozdíl mezi hodnotovými třídami a strukturami spočívá ve viditelnosti jejich datových členů: zatímco členy hodnotových tříd jsou implicitně soukromé, členy hodnotových struktur disponují výchozím veřejným přístupem.

```
// Definice hodnotové třídy.
public value class Vektor1
{
   int x, y, z;
};

// Definice hodnotové struktury.
public value struct Vektor2
{
   int x, y, z;
};
```

Představená hodnotová třída i hodnotová struktura jsou zjednodušenými modely matematického vektoru. Obě entity ve svých tělech deklarují tři celočíselné proměnné typu `int`, jež reprezentují složky vektoru ve trojrozměrném prostoru. Instanci hodnotové třídy i struktury můžeme alokovat na zásobníku:

```
// Instanciace hodnotové třídy.
Vektor1 v1;
// Tyto příkazy nebudou moci být provedeny vzhledem k tomu, že
// datové položky hodnotové třídy nejsou pro vnější kód viditelné.
v1.x = 1;  v1.y = 2;  v1.z = 3;

// Instanciace hodnotové struktury.
Vektor2 v2;
// Tohle je v pořádku, neboť datové položky struktury jsou
// veřejně přístupné.
v2.x = 0;  v2.y = 1;  v2.z = 2;
```

Hodnotové třídy ani hodnotové struktury nesmějí vystupovat jako bázové třídy, od nichž by mohly být v procesu jednoduché dědičnosti odvozovány další podtřídy. Ačkoliv není možné od jedné hodnotové třídy nebo struktury odvodit jinou programovou entitu, hodnotové třídy i struktury mohou implementovat jedno nebo i několik řízených rozhraní. Řízená rozhraní se v jazyce C++/CLI definují pomocí klíčových slov `interface class`. Když budeme chtít vytvořit řízené rozhraní, použijeme zmíněná klíčová slova, přičemž do těla rozhraní umístíme prototypy datových členů.

```
interface class IVektor
{
   void VypocitatSkalarniSoucin();
   void VypocitatVektorovySoucin();
};
```

Rozhraní `IVektor` je řízeným rozhraním, které obsahuje deklarace dvou metod. Kdybychom chtěli, aby výše napsaná hodnotová třída `Vektor1` zavedla implementaci rozhraní `IVektor`, počínali bychom si takto:

```
public value class Vektor1 : public IVektor
{
   int x, y, z;
public:
   virtual void VypocitatSkalarniSoucin() {}
   virtual void VypocitatVektorovySoucin() {}
};
```

V okamžiku, kdy se hodnotová třída `Vektor1` rozhodne implementovat rozhraní `IVektor`, musí zavést definice všech datových členů, jejichž prototypy rozhraní deklaruje.

TIP	
	Řízené rozhraní mohou vývojáři v jazyce C++/CLI zakládat také pomocí klíčových slov `interface struct`. Kontextově senzitivní slovní spojení `interface struct` je tedy ekvivalentem příkazu `interface class`.

Hodnotovou třídu popřípadě strukturu jsme mohli v C++ s Managed Extensions vytvořit na různých paměťových působištích, k nimž patřil zásobník, nativní hromada a řízená hromada. Standardně jsou instance hodnotových datových typů ukládány v zásobníku, ovšem pomocí operátoru `__nogc new` bylo možné založit instanci hodnotové třídy nebo struktury také v nativní, tedy neřízené hromadě C++. Dejme tomu, že bychom měli jednoduchou hodnotovou strukturu `Bod`, jejíž instance budou reprezentovat body v rovině. V jazyce C++ s Managed Extensions by definice takovéto struktury byla následovní:

```
// Definice hodnotové struktury jazyka C++ s Managed Extensions.
public __value struct Bod
{
   Int32 x, y;
};
```

Kdybychom chtěli alokovat instanci struktury `Bod` v nativní hromadě, postupovali bychom takto:

```
// Instanciace struktury.
Bod __nogc * bod1 = __nogc new Bod();
// Inicializace datových položek instance struktury.
bod1->x = 2;
bod1->y = 4;
// Zobrazení informací o souřadnicích bodu.
MessageBox::Show(String::Concat(S"Souřadnice bodu: [",
(bod1->x).ToString(), S",", (bod1->y).ToString(), S"]."),
S"Alokace struktury v nativní hromadě", MessageBoxButtons::OK,
MessageBoxIcon::Information);
// Likvidace instance struktury.
delete bod1;
```

V C++/CLI neexistuje žádný přímý protějšek operátoru `__nogc new`, což znamená, že v tomto prostředí nemůžeme založit instanci hodnotové třídy nebo struktury v nativní hromadě C++.

V obou programovacích jazycích je možné umístit instanci hodnotové třídy nebo struktury na řízenou hromadu. To se děje kupříkladu tehdy, když využijeme služeb mechanizmu sjednocení typů a instanci hodnotové struktury `Bod` vložíme do objektové skříňky, domovem které je řízená hromada. Takto situace vypadá v jazyce C++ s Managed Extensions:

```
Bod bod1;
bod1.x = 10; bod1.y = 20;
System::Object __gc* obj = __box(bod1);
```

A takhle zase v C++/CLI:

```
Bod bod1;
bod1.x = 10; bod1.y = 20;
System::Object^ obj = bod1;
```

Podíváte-li se na oba výpisy zdrojového kódu, ihned rozeznáte, že jedna instance struktury `Bod` bude uložena na zásobník, zatímco její „klon" najde své místo v objektové skříňce na řízené hromadě.

Instance hodnotového datového typu smí být na řízenou hromadu umístěna také v případě, že bude uložena v řízeném poli. Řízená pole se v jazyce C++/CLI nedeklarují pomocí symbolu `__gc[]` a rovněž nedochází k jejich dynamické instanciaci prostřednictvím operátoru `__gc new`. Nově se řízená pole deklarují příkazem `array`, přičemž jejich zakládání má na starosti operátor `gcnew`. Zkusme nyní napsat kód, jenž sestrojí řízené pole o deseti prvcích, z nichž každý bude naplněn instancí hodnotové struktury `Bod`. Výsledek naší práce by mohl mít třeba následující podobu:

```
// Deklarace a okamžitá instanciace dynamického řízeného pole.
array<Bod>^ pole = gcnew array<Bod>(10);
// Inicializace datových položek hodnotových struktur, jež jsou
// uloženy v řízeném poli.
pole[0].x = 5;
pole[0].y = 7;
pole[9].x = 10;
pole[9].y = 13;
```

Na prvním řádku zakládáme nové, 10prvkové pole, které může být osazeno stejným počtem instancí hodnotové struktury. Všimněte si, že operátor `gcnew` nám navrací manipulátor (`^`) na vzniklé pole. Pomocí tohoto manipulátoru můžeme pole kdykoliv na řízené hromadě najít, a to i tehdy, bude-li z důvodu práce automatického správce paměti přemístěno na novou pozici. Jakmile je pole na světě, můžeme inicializovat instance hodnotové struktury `Bod`, které tvoří obsahovou výplň jeho jednotlivých prvků. V naší ukázce nastavujeme souřadnice prvního (index 0) a posledního (index 9) bodu čili instance zvolené struktury. Společně s polem jsou na řízené hromadě alokované rovněž tyto instance.

Manipulátor, který obdržíme, nám umožňuje identifikovat celé pole – pole se tedy tváří jako jeden komplexní a víceméně monolitní objekt. Jestliže budeme chtít přistupovat přímo k datovým členům instancí hodnotové struktury, můžeme aplikovat dvě postupové varianty. Tu první můžete vidět v zobrazeném fragmentu zdrojového kódu: pomocí symbolu `[]` analyzujeme kýžené prvky pole, vybíráme cílové instance a provádíme s nimi zamýšlené akce. Druhá alternativa se spoléhá na využití tzv. interních ukazatelů. Díky

interním ukazatelům se můžeme dotázat na konkrétní prvek pole a ne na pole jako celek. Interní ukazatele se v jazyce C++/CLI vytvářejí příkazem `interior_ptr`. Jestliže bychom chtěli pomocí interního ukazatele pozměnit souřadnice desátého bodu, jenž je uložen na pozici `pole[9]`, postupovali bychom takto:

```
interior_ptr<Int32> ip = &pole[9].y;
*ip = 20;
MessageBox::Show("Souřadnice 10. bodu: [" + pole[9].x +
   "," + pole[9].y + "].", "Použití interního ukazatele",
   MessageBoxButtons::OK, MessageBoxIcon::Information);
```

Při deklaraci interního ukazatele musíme stanovit, na co bude vlastně nasměrován – v našem případě bude ukazatel mířit na 32bitovou celočíselnou hodnotu typu `Int32`. Poté, co je interní ukazatel náležitě deklarován, inicializujeme jej za asistence referenčního operátoru (`&`). Pak aplikujeme operátor pro dereferenci (`*`), pomocí kterého modifikujeme hodnotu datové položky instance hodnotové struktury.

Interní ukazatel nám posloužil pro přístup k datové položce instance hodnotové struktury. Budete-li mít chuť, můžete interní ukazatel nasměrovat také na samotnou instanci hodnotové struktury a ne na specifickou datovou položku. Tuto situaci simuluje další fragment programového kódu:

```
interior_ptr<Bod> ipb = &pole[9];
ipb->x++;
ipb->y++;
MessageBox::Show("Souřadnice 10. bodu: [" + pole[9].x +
   "," + pole[9].y + "].", "Použití interního ukazatele",
   MessageBoxButtons::OK, MessageBoxIcon::Information);
```

Místo toho, aby interní ukazatel směroval na datovou položku typu `Int32`, míří na celou instanci hodnotové struktury `Bod`, která se nachází na poslední pozici řízeného pole. Máme-li k dispozici ukazatel na instanci, můžeme zapsat přístupový operátor (`->`) a inkrementovat hodnoty obou datových položek.

Programátoři pracující v jazyce C++ s Managed Extensions mohli do těl hodnotových tříd a struktur vkládat definice implicitních instančních konstruktorů.

```
// Hodnotové struktury a třídy definované v C++ s Managed Extensions
// podporovaly implicitní instanční konstruktory.
public __value struct Bod
{
   Int32 x, y;
   Bod()
   {
      Console::WriteLine(S"Výchozí konstruktor.");
   }
};
```

To již neplatí v C++/CLI, poněvadž zde jsou implicitní instanční konstruktory hodnotových tříd a struktur zakázány. Tvůrci jazyka se k tomuto kroku odhodlali z důvodu chování běhového prostředí CLR, které nezaručovalo stoprocentní spolehlivost při aktivaci implicitních instančních konstruktorů u instancí hodnotových tříd a struktur.

```
// C++/CLI: Implicitní konstruktory nelze použít.
public value struct Bod
{
   Int32 x, y;
   Bod()
   {
      // Zde by kompilátor ohlásil chybu.
      Console::WriteLine("Výchozí konstruktor.");
   }
};
```

Odkazové (řízené) třídy a struktury

Stěžejním pilířem jazykové specifikace C++/CLI jsou odkazové neboli řízené třídy a struktury. Instance těchto programových entit mohou v celé své šíři těžit ze všech kladných vlastností objektově orientovaného programování na platformě Microsoft .NET Framework 2.0. V jazyce C++ s Managed Extensions jsme řízené třídy a struktury definovali příkazy `__gc class` a `__gc struct`. Jazyk C++/CLI dřívější klíčová slovní spojení nahrazuje novými kandidáty `ref class` a `ref struct`. Instance odkazových tříd a struktur se nyní zakládají prostřednictvím operátoru `gcnew`, jenž nahrazuje dřívější operátor `__gc new`. Všechny sestrojené instance jsou implicitně ukládány na řízenou hromadu, kde jsou pod dohledem automatického správce paměti. Po pravdě řečeno, objekty odkazových tříd a struktur ani nemohou sídlit jinde – řízená hromada je jediným místem v operační paměti počítače, kde se mohou tyto entity nacházet.

Tematiku řízených tříd zahájíme představením třídy `BublinoveOkno`, jejíž instance nám pomohou se zobrazováním bublinových oken bledožluté barvy, které budou vystupovat z oznamovací oblasti hlavního panelu systému Windows.

```
using namespace System;
using namespace System::Windows::Forms;
using namespace System::Drawing;

public ref class BublinoveOkno
{
private:
   ::NotifyIcon^ OznamovaciOblast;
   ::Icon^ Ikona;
   String^ HlavniText, ^TextVTitulku;
public:
   BublinoveOkno()
   {
      this->OznamovaciOblast = gcnew NotifyIcon();
      this->OznamovaciOblast->Icon = gcnew Icon("d:\\directx.ico");
      this->HlavniText = "Programovací jazyk C++/CLI je plný nových " +
      "a vzrušujících věcí!";
      this->TextVTitulku = "Máte rádi C++? Přidejte se k nám!";
      this->OznamovaciOblast->Visible = true;
      this->OznamovaciOblast->BalloonTipTitle =
         this->TextVTitulku;
      this->OznamovaciOblast->BalloonTipText =
         this->HlavniText;
      this->OznamovaciOblast->ShowBalloonTip(10000, TextVTitulku,
         HlavniText, ToolTipIcon::Info);
   }
};
```

Máte-li zkušenosti s jazykem C++ s Managed Extensions, tak jedinou podstatnou změnou, kterou je nutno mít na paměti, je použití nových kontextových klíčových slov `ref class` v hlavičce řízené třídy.

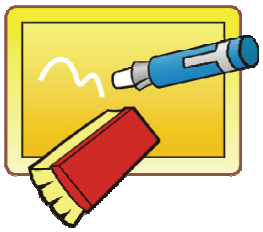
V soukromé části třídy deklarujeme čtyři datové položky, jejichž smysl tkví v konfiguraci oznamovací oblasti jakožto i samotného bublinového okna. Ještě předtím, než můžeme bublinové okno zobrazit, musíme do oznamovací oblasti hlavního panelu vložit ikonu, s kterou bude bublinové okno asociováno. Hlavní roli v tomto procesu hraje instance třídy `NotifyIcon` z jmenného prostoru `System::Windows::Forms`. Pro určení ikony využíváme služeb instance třídy `Icon` z prostoru jmen `System::Drawing`. Dále již nastavujeme potřebné vlastnosti a nakonec voláme metodu `ShowBalloonTip`, která na základě námi dodaných vstupních argumentů zabezpečí sestavení a zobrazení bublinového okna.

Instanci naší řízené třídy založíme pomocí operátoru `gcnew`:

```
BublinoveOkno^ Okno = gcnew BublinoveOkno();
```

Tento syntaktický zápis způsobí alokaci nové instance třídy `BublinoveOkno` v nulté generaci řízené hromady.

POZNÁMKA

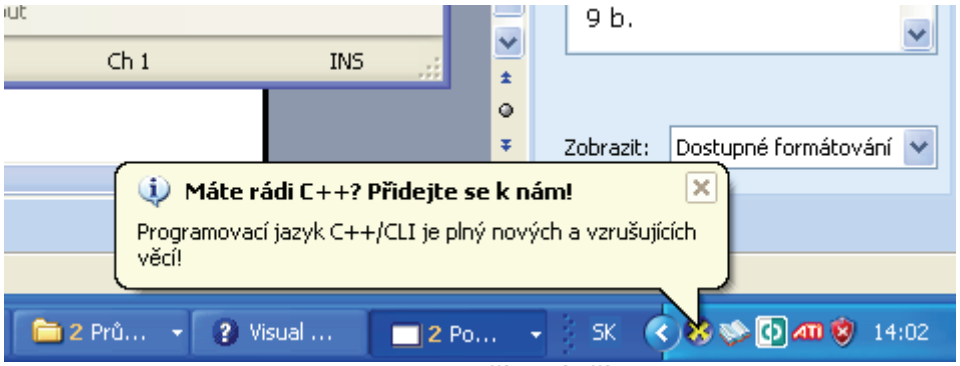


O skutečnosti, že zrozená instance je doopravdy vytvořena v generaci č. 0, vás přesvědčí například níže uvedený zdrojový kód:

```
BublinoveOkno^ Okno = gcnew BublinoveOkno();
Int32 Generace = GC::GetGeneration(Okno);
MessageBox::Show("Objekt je uložen v generaci č. " +
   Generace + ".", "Určení objektové generace",
   MessageBoxButtons::OK, MessageBoxIcon::Information);
```

Když operátor `gcnew` dokončí vytváření nové objektové instance, vrátí sledovací manipulátor na tuto instanci. Tento manipulátor plní ve vztahu k instancím řízených tříd stejnou funkci jako standardní ukazatel vůči nativním objektům. Jinými slovy, pomocí manipulátoru je řízená instance v kterémkoliv okamžiku jednoznačně identifikovatelná a dosažitelná. Jak vyplývá z jeho názvu, sledovací manipulátor dovede najít řízenou instanci, ať už se nachází na libovolném konci řízené hromady. Jelikož objekty spočívající na řízené hromadě mohou být přesouvány, mění se také jejich paměťové adresy. To však pro nás nepředstavuje žádný problém, neboť běhové prostředí CLR v případě potřeby zcela automaticky aktualizuje odpovídající manipulátor. Výsledkem je, že manipulátor `^` nikdy neztratí svou cílovou instanci.

Když kód spustíte, spatříte bublinové okno, jehož podobu zachytává obr. 2.3.



Obr. 2.3: Instance řízené třídy v akci

Přicházíte-li z C++ s Managed Extensions, pak si můžete manipulátor ^ představit jako nástupce řízeného ukazatele `__gc*`. Sledovací manipulátor může ukazovat pouze na celý objekt, nikoliv na jednotlivé datové části objektu. (Pokud potřebujete ukazatel na zapouzdřený objekt, můžete použít interní ukazatel.)

V okamžiku, kdy provedete deklaraci sledovacího manipulátoru, kompilátor jazyka C++/CLI jej neprodleně inicializuje hodnotou nulové konstanty, kterou reprezentuje klíčové slovo `nullptr`. Pravdivost tvrzení uvedeného v předcházející větě prokazuje další výpis zdrojového kódu:

```
BublinoveOkno^ obj;
if (obj == nullptr)
   Console::WriteLine("Manipulátor obsahuje nulovou hodnotu.");
else
   Console::WriteLine("Manipulátor neobsahuje nulovou hodnotu.");
Console::Read();
```

Z informační zprávy vypsané v okně konzoly se dovíme, že v odkazové proměnné `obj` se nachází `nullptr`, tedy nulová hodnota (nebo jinak řečeno, manipulátor ^, který není nikam nasměrován).

Sledovací manipulátor můžeme dereferencovat a nepřímo tak zavolat metodu instance řízené třídy. Níže je uvedena třída `DatovyProud`, která definuje jednu metodu pro načtení řetězců z textových souborů.

```
using namespace System::IO;

public ref class DatovyProud
{
public:
   String^ NacistTextZeSouboru(String^ Soubor)
   {
      StreamReader^ Text = gcnew StreamReader(Soubor);
      String^ NactenyText = Text->ReadToEnd();
      return NactenyText;
   }
};
```

Když založíme instanci této třídy, můžeme použít manipulátor ^ pro implicitní aktivaci veřejně přístupné instanční metody.

```
DatovyProud^ dat = gcnew DatovyProud();
// Sledovací manipulátor můžeme podrobit dereferenci stejně jako
// standardní ukazatel na instanci nativní třídy.
String^ text = (*dat).NacistTextZeSouboru("d:\\TextovySoubor.txt");
```

```
Console::WriteLine(text);
Console::Read();
```

Novinkou jazyka C++/CLI je možnost využít pro instanciaci řízené třídy zásobníkovou sémantiku. To znamená, že instanci třídy můžeme vytvořit podobně, jako zakládáme instance hodnotových tříd a struktur.

```
// Instanci odkazové třídy vytváříme jako instanci hodnotové třídy.
DatovyProud dat;
Console::WriteLine(dat.NacistTextZeSouboru("d:\\TextovySoubor.txt"));
Console::Read();
```

Nicméně, je docela dost možné, že ve vás nová syntaktická forma vzbudí interesantní otázku: Dobrá, znamená to tedy, že vývojáři mohou v C++/CLI alokovat instance odkazových tříd na zásobníku? Jedná se o docela pozoruhodnou programátorskou hříčku, nemyslíte? Ať tak či onak, pokusíme se vás ještě chvíli držet v napětí. Abychom rozptýlili možné počáteční pochybnosti, tak učiňme prohlášení, že veškerý uvedený zdrojový kód je zcela korektní a kompilátor jazyka C++/CLI nebude mít s jeho překladem žádné potíže. Pokud však budeme chtít přijít všemu na kloub, musíme se ponořit do hlubin jazyka MSIL. Po nahlédnutí do příslušného fragmentu kódu MSIL zjistíme toto:

```
.method assembly static int32  main(string[] args) cil managed
{
  // Code size       32 (0x20)
  .maxstack  2
  .locals ([0] class DatovyProud dat)
  IL_0000:  ldnull
  IL_0001:  stloc.0
  IL_0002:  newobj     instance void DatovyProud::.ctor()
  IL_0007:  stloc.0
  IL_0008:  ldloc.0
  IL_0009:  ldstr      "d:\\TextovySoubor.txt"
  IL_000e:  call       instance string
            DatovyProud::NacistTextZeSouboru(string)
  IL_0013:  call       void [mscorlib]System.Console::WriteLine(string)
  IL_0018:  call       int32 [mscorlib]System.Console::Read()
  IL_001d:  pop
  IL_001e:  ldc.i4.0
  IL_001f:  ret
} // end of method 'Global Functions'::main
```

Na řádku, jenž obsahuje návěstí `IL_0002`, se nachází příkaz `newobj instance void DatovyProud::.ctor()`. Co to pro nás znamená? Nuže to, že kompilátor emituje příkaz `newobj`, což je nezvratný důkaz, jenž dokládá průběh těchto operací:

1. Na řízené hromadě běhového prostředí CLR je alokován prostor pro novou instanci a tato instance je záhy vytvořena.

2. Je volán instanční konstruktor (metoda s názvem `.ctor`) vzniklé instance. Vzhledem k tomu, že třída `DatovyProud` neobsahuje nic, co by vypadlo jako instanční konstruktor, kompilátor jazyka C++/CLI generuje implicitní instanční konstruktor, v jehož těle je volán konstruktor primární bázové třídy `System::Object` (čemuž odpovídá MSIL instrukce `call instance void [mscorlib]System.Object::.ctor()`). Běhové prostředí CLR tedy aktivuje implicitní instanční konstruktor.

Závěr naší detektivní práce je jednoznačný: Přestože jsme pro alokaci instance odkazové třídy použili zásobníkovou sémantiku, kompilátor generuje instrukce, které zaručí založení této instance na řízené hromadě.

Generování instancí řízených tříd pomocí zásobníkové sémantiky má jeden důležitý důsledek, který je nutno brát v potaz. Pokud řízená třída definuje destruktor, tak ten je zavolán poté, co se odkazová proměnná (uložena na zásobníku a třímající referenci na objekt dané třídy) ocitne mimo svého oboru platnosti.

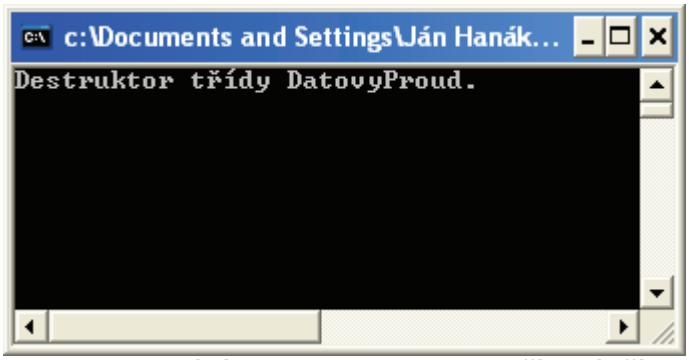
Abychom se obeznámili s praktickou aplikací, obohaťme naši třídu `DatovyProud` o destruktor:

```
using namespace System::IO;

public ref class DatovyProud
{
public:
   String^ NacistTextZeSouboru(String^ Soubor)
   {
      StreamReader^ Text = gcnew StreamReader(Soubor);
      String^ NactenyText = Text->ReadToEnd();
      return NactenyText;
   }
   // Třída je opatřena destruktorem.
   ~DatovyProud()
   {
      Console::WriteLine("Destruktor třídy DatovyProud.");
   }
};
```

Instanci třídy vytvoříme podle zásobníkové sémantiky, přičemž instanciační příkaz umístíme do samostatného programového bloku:

```
int main(array<System::String ^> ^args)
{
   {
      DatovyProud dat;
   }
   Console::Read();
}
```

Po přeložení a spuštění aplikace spatříme následující informační zprávu:



Obr. 2.4: Volání destruktoru instance řízené třídy

Běhové prostředí CLR volá destruktor instance založené pomocí zásobníkové sémantiky, jakmile tato překročí svůj obor působnosti.

Patříte-li mezi zvídavé vývojáře, pak pro vás máme další programátorskou pozoruhodnost. Totiž podobně, jak je možné zakládat instance odkazových typů jakoby šlo o typy hodnotové, je rovněž dovoleno vytvářet instance hodnotových typů pomocí objektové sémantiky. Řečeno jinými slovy, instance hodnotových typů smíme zakládat také použitím operátoru `gcnew`. Povězme, že do editoru zdrojového kódu Visual C++ 2005 zapíšeme tento řádek programového kódu:

```
Int32^ i = gcnew Int32();
```

První změnou je skutečnost, že kompilátor vás nezastaví a nevypíše chybovou zprávu (kompilátor jazyka C++ s Managed Extensions tak shovívavý ani zdaleka není). Nestěžuje-li si kompilátor, pak je příkaz ze syntaktického hlediska naprosto v pořádku. Dobře, ale co vlastně dělá? Poněvadž je aplikován operátor `gcnew`, můžeme se domnívat, že je založena instance systémového hodnotového typu `Int32` na řízené hromadě, co říkáte? Bez ohledu na to, co si asi můžeme myslet, je situace poněkud složitější. Je to proto, že instance hodnotového typu `Int32` nemůže být „jen tak" umístěna na řízenou hromadu. Uvedený příkaz ji tam však přesto uloží. Pokud se ptáte, jak je to možné, rádi bychom vám připomněli, že běhové prostředí CLR disponuje mechanizmem sjednocení typů. A právě tento mechanizmus je odpovědný za to, že tento příkaz bude fungovat. Důsledkem je, že na řízené hromadě bude sestrojena objektové skříňka třídy `System::ValueType`, do které bude vložena instance hodnotového typu `Int32`.

To, co jsme si řekli, dokládá následující fragment zdrojového kódu jazyka MSIL:

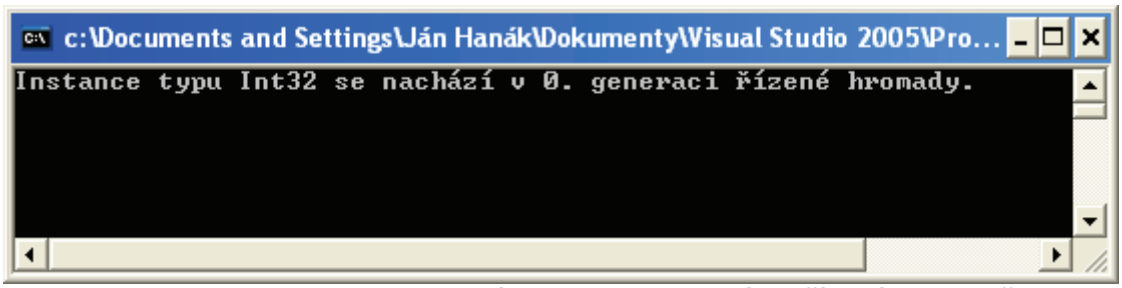
```
.method assembly static int32  main(string[] args) cil managed
{
  // Code size       11 (0xb)
  .maxstack  1
  .locals ([0] class [mscorlib]System.ValueType
modopt([mscorlib]System.Int32)
modopt([mscorlib]System.Runtime.CompilerServices.IsBoxed) i)
  IL_0000:  ldnull
  IL_0001:  stloc.0
  IL_0002:  ldc.i4.0
  IL_0003:  box        [mscorlib]System.Int32
  IL_0008:  stloc.0
  IL_0009:  ldc.i4.0
  IL_000a:  ret
} // end of method 'Global Functions'::main
```

Program iniciuje sestrojení instance třídy `System.ValueType` s příznakem `modopt([mscorlib]System.Runtime.CompilerServices.IsBoxed)`. Po uskutečnění počátečních operací je zpracována instrukce `box [mscorlib]System.Int32`, která provádí zabalení instance typu `Int32` a vkládá ji do objektové skříňky na řízené hromadě.

Pokud je instance typu `Int32` situována na řízené hromadě, můžeme zjistit, ve které generace se skutečně nachází:

```
Int32^ i = gcnew Int32();
Console::WriteLine("Instance typu Int32 se nachází v " +
   GC::GetGeneration(i) + ". generaci řízené hromady.");
Console::Read();
```

Výsledek můžete vidět na obr. 2.5.



Obr. 2.5: Instance hodnotového typu alokovaná na řízené hromadě

Destruktory a finalizační metody v C++/CLI

Jste-li zručnými programátory v jazyce C++ s Managed Extensions, pak vám jistě není třeba připomínat vztah mezi destruktory a finalizačními metodami. Když jste v tomto prostředí do těla třídy vložili definici destruktoru, kompilátor jazyka C++ s Managed Extensions ji interně převedl do podoby finalizační metody. Finalizační metodu volal automatický správce paměti v procesu nedeterministické dealokace instance třídy. Jako protiváha k časově nestálé finalizační metodě se jevila metoda `Dispose` rozhraní `IDisposable`. Tuto metodu jsme mohli zavést do těla třídy a naprogramovat tak úklidové operace, jež měly být spuštěny v přesně vymezeném časovém okamžiku. V C++ s Managed Extensions vývojáři rozlišovali mezi implicitním uvolněním objektových zdrojů pomocí finalizéru a explicitní dealokací prostřednictvím metody `Dispose`. Rozhodnete-li se přejít na jazyk C++/CLI, musíte se připravit na malé zemětřesení, neboť v oblasti destruktorů a finalizačních metod nezůstal takříkajíc kámen na kameni.

Destruktor != finalizační metoda

V C++/CLI již není destruktor kompilátorem automaticky transformován do formy finalizační metody. Abychom si tuto skutečnost názorně demonstrovali, připravili jsme si jednoduchou odkazovou třídu, jejíž programový kód je takovýto:

```
public ref class C
{
public:
   C()
   {
      Console::WriteLine("Instanční konstruktor třídy C.");
   }
   ~C()
   {
      Console::WriteLine("Destruktor třídy C.");
   }
};
```

Tato třída se jmenuje `C` a obsahuje konstruktor i destruktor. Když kód třídy přeložíme a otevřeme řízený modul v programu IL DASM, zjistíme, že kompilátor vygeneroval v návaznosti na námi definovaný destruktor tyto metody:

- `Dispose : void(bool),`
- `Dispose : void(),`
- `~C : void().`

Podívejme se na tyto metody blíže. Poslední metoda představuje destruktor, přičemž v jejím těle se nacházejí instrukce jazyka MSIL, jež odpovídají kódu C++/CLI, který jsme do těla destruktoru třídy `C` vložili. Obraz metody `~C : void()` je proto následovní:

```
.method private hidebysig instance void  '~C'() cil managed
{
  // Code size       11 (0xb)
  .maxstack  1
  IL_0000:  ldstr      bytearray (44 00 65 00 73 00 74 00 72 00 75 00 6B 00
                       74 00 // D.e.s.t.r.u.k.t.
                       6F 00 72 00 20 00 74 00 59 01 ED 00 64 00 79 00
                       // o.r. .t.Y...d.y.
                       20 00 43 00 2E 00 ) //  .C...
  IL_0005:  call       void [mscorlib]System.Console::WriteLine(string)
  IL_000a:  ret
} // end of method C::'~C'
```

V destruktoru se neodehrává nic neobyčejného: dochází k načtení textového řetězce na zásobník a zavolání metody `WriteLine` třídy `Console` z jmenného prostoru `System`. Příkaz `ret` působení destruktoru ukončuje.

Nyní se zaměřme na metodu `Dispose`. Jak jste již pravděpodobně zaregistrovali, kompilátor jazyka C++/CLI sestavil dvojici metod s názvem `Dispose`, což je neklamný důkaz toho, že máme do činění s přetíženou metodou. A vskutku – zatímco jedna verze je prosta formálních parametrů, v signatuře další se vyjímá jeden parametr typu `bool`. Zaměřme se nejdříve na metodu `Dispose` s prázdným seznamem parametrů. Její obraz v jazyce MSIL je takovýto:

```
.method public hidebysig virtual final instance void
        Dispose() cil managed
{
  // Code size       14 (0xe)
  .maxstack  2
  IL_0000:  ldarg.0
  IL_0001:  ldc.i4.1
  IL_0002:  callvirt   instance void C::Dispose(bool)
  IL_0007:  ldarg.0
  IL_0008:  call       void [mscorlib]System.GC::SuppressFinalize(object)
  IL_000d:  ret
} // end of method C::Dispose
```

Z kódu vyčítáváme dvě důležité věci:

1. Instrukce `callvirt` aktivuje přetíženou verzi metody `Dispose` (`C::Dispose(bool)`).

2. Instrukce `call` iniciuje spuštění statické metody `SuppressFinalize` třídy `GC`. Tato metoda zabraňuje spuštění finalizační metody pro aktuální instanci třídy.

Zůstává nám již pouze poslední metoda se signaturou `Dispose : void(bool)`. Nuže, sem s ní:

```
.method family hidebysig newslot virtual
        instance void  Dispose(bool  marshal( unsigned int8) A_1) cil
managed
{
  // Code size       18 (0x12)
  .maxstack  1
```

```
  IL_0000:  ldarg.1
  IL_0001:  brfalse.s  IL_000b
  IL_0003:  ldarg.0
  IL_0004:  call       instance void C::'~C'()
  IL_0009:  br.s       IL_0011
  IL_000b:  ldarg.0
  IL_000c:  call   instance void [mscorlib]System.Object::Finalize()
  IL_0011:  ret
} // end of method C::Dispose
```

Parametrická virtuální metoda `Dispose` přebírá argument nativního datového typu `unsigned int8`, jenž je konvertován na hodnotu typu `bool`. Jestliže je dodaným argumentem logická hodnota `true`, pak je volán destruktor instance třídy (jak praví instrukce `call instance void C::'~C'()`). V opačném případě (je-li dodána hodnota `false`) je zpracována instrukce `brfalse.s IL_000b`, která další běh programu směruje na řádek s návěstím `IL_000b` a potažmo `IL_000c`, kde je volána metoda `Finalize` primární bázové třídy `System::Object`.

V oblasti destruktorů se mezi jazyky C++ s Managed Extensions změnilo opravdu hodně. Zatímco v prvně jmenovaném jazyce byl destruktor převlečenou finalizační metodou `Finalize`, nový nástupce C++/CLI tuto koncepci boural. Když totiž v těle řízené třídy jazyka C++/CLI napíšete destruktor, kompilátor sestaví tři metody: jeden opravdový destruktor a dvě verze přetížené metody `Dispose`. Kromě toho je deklarace vámi definované třídy zcela automaticky rozšířena o implementaci rozhraní `IDisposable`. Všechny zmíněné indicie nás vedou k tomu, že destruktor je v C++/CLI mapován na metodu `Dispose`. Destruktor je po novém představitelem deterministické metody, která smí být volána z klientského programového kódu, a která se může pochlubit předvídatelným okamžikem své exekuce. Když na instanci odkazové třídy použijeme operátor `delete` (podobně, jako jsme to dělávali v nativním C++ s objekty alokovanými na standardní hromadě C++), kompilátor bude aktivovat metodu `Dispose`.

```
// Založení instance odkazové třídy C.
C^ obj_c = gcnew C();
// Použití operátoru delete v souvislosti s instancí odkazové třídy
// způsobí volání metody Dispose.
delete obj_c;
```

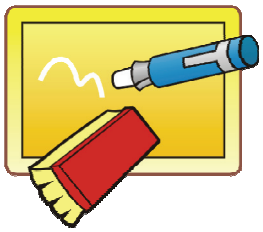
A zde je útržek předmětného kódu jazyka MSIL:

```
.method assembly static int32  main(string[] args) cil managed
{
  // Code size       27 (0x1b)
  .maxstack  1
  .locals ([0] class [mscorlib]System.IDisposable V_0,
           [1] class C obj_c,
           [2] int32 V_2)
  IL_0000:  ldnull
  IL_0001:  stloc.1
  IL_0002:  newobj     instance void C::.ctor()
  IL_0007:  stloc.1
  IL_0008:  ldloc.1
  IL_0009:  stloc.0
  IL_000a:  ldloc.0
  IL_000b:  brfalse.s  IL_0017
  IL_000d:  ldloc.0
  IL_000e:  callvirt instance void [mscorlib]System.IDisposable::Dispose()
  IL_0013:  ldc.i4.0
  IL_0014:  stloc.2
```

```
  IL_0015:  br.s       IL_0019
  IL_0017:  ldc.i4.0
  IL_0018:  stloc.2
  IL_0019:  ldc.i4.0
  IL_001a:  ret
} // end of method 'Global Functions'::main
```

Zaměřte se prosím na řádek s návěstím `IL_000e`. Zde dochází k volání bezparametrické metody `Dispose`. Když je tato metoda zavolána, začne se zpracovávat její programový kód. Tento kód dělá dvě podstatné věci: jednak spouští parametrickou verzi metody `Dispose` a jednak probouzí k životu statickou metodu `SuppressFinalize` třídy `System::GC`, která potlačuje průběh finalizačního procesu. Pro programátory v C++/CLI je ovšem důležité sdělení, že pokud bude metodě `Dispose(bool)` předána logická hodnota `true`, bude zavolán destruktor instance řízené třídy. Destruktor v MSIL je v tomto ponímání přímým ekvivalentem destruktoru v C++/CLI, což znamená, že příkazy, které zapíšeme v C++/CLI, budou provedeny při volání destruktoru z těla přetížené metody `Dispose`.

POZNÁMKA



Destruktor může být zavolán také explicitně. V C++/CLI je přímé volání destruktoru funkčně shodné s použitím operátoru `delete`.

```
// Aktivace destruktoru přes operátor delete...
C^ obj_c = gcnew C();
delete obj_c;

// ...je ekvivalentní explicitnímu zavolání destruktoru.
C^ obj_c = gcnew C();
obj_c->~C();
```

V obou uvedených případech bude kompilátorem jazyka C++/CLI sestaven stejný MSIL kód se třemi metodami (destruktorem a dvěma variantami metody `Dispose`).

Nový pracovní postup destruktorů potěší zejména ty programátory, kteří se jen velice těžko sžívali s poněkud svévolným chováním destruktorů v C++ s Managed Extensions. Je nasnadě, že tímto počinem učinili tvůrci řízeného C++ veliký krok kupředu.

Finalizér a finalizační metoda

Leaderem nedeterministické finalizace se stává nově uváděná metoda, která se nazývá finalizér. Tato metoda však již není přímým protějškem destruktoru, jak tomu bylo v C++ s Managed Extensions. Nyní je finalizér syntakticky i sémanticky oddělenou jednotkou, která hraje roli při implicitní dealokaci objektových prostředků pomocí automatického správce paměti. Finalizér je tedy jakási „poslední záchrana", která je schopna uvolnit nativní objektové zdroje těsně předtím, než bude objekt dealokován automatickým správcem paměti. Finalizér se stává požadovaným v těch situacích, kdy nebyl zpracován destruktor instance (a obsazené zdroje nebyly ještě uvolněny).

UPOZORNĚNÍ	
	Z terminologického hlediska musíme v jazyce C++/CLI rozlišovat mezi pojmy finalizér a finalizační metoda. Jak se za několik chvil dozvíte, finalizér je dosud neznámou metodou, která není totožná s metodou `Finalize`.

Mezi destruktorem a finalizérem existuje jistý a nutno říci, že docela úzký vztah. Pokud odkazová třída definovaná v C++/CLI požaduje vyhrazení systémových prostředků, měla by také definovat destruktor, ve kterém dojde k explicitní, a tudíž deterministické destrukci těchto zdrojů. Třída, která se rozhodne implementovat finalizér, by měla být rovněž opatřena destruktorem, aby bylo možné alokované jednotky paměti uvolnit pokud možno v předstihu. Za těchto okolností však není cílem duplikovat kód – místo toho by se měl destruktor pokusit dealokovat co nejvíc prostředků a zbytek „finalizačního" kódu by měl být vložen do finalizéru.

Mnozí programátoři se mohou ptát: Dobrá, když uvolním všechny neřízené zdroje v destruktoru, nač mám ještě psát finalizér? Stručná odpověď zní: je to podmíněné inovovaným dealokačním scénářem. Když vložíme do třídy definici destruktoru, kompilátor jej na nízké úrovni namapuje na metodu `Dispose`. Tato metoda je volána při aktivaci destruktoru přes operátor `delete`. Co se však stane v případě, že destruktor nebude zavolán? Pak se v zájmu uvolnění vyhrazených zdrojů musíme chtě nechtě spolehnout na finalizér, který provede implicitní dealokaci kýžených prostředků. Finalizér bude automaticky zavolán automatickým správcem paměti při likvidaci objektu na řízené hromadě.

Jelikož destruktor třídy již není v C++/CLI synonymem pro finalizér, musíte se obeznámit s novou syntaxí, jejíž prostřednictvím se finalizační metoda definuje. Fragment zdrojového kódu uvedený níže předvádí použití finalizéru uvnitř odkazové třídy:

```
public ref class D
{
public:
   D()
   {
      Console::WriteLine("Instanční konstruktor třídy.");
   }
   !D()
   {
      Console::WriteLine("Finalizér třídy.");
   }
};
```

Ze syntaktického hlediska je finalizér metodou se stejným názvem jako třída, přičemž samotný název předchází symbol vykřičníku (`!`). Finalizéru není dovoleno deklarovat formální parametry a stejně tak nesmí mít žádnou návratovou hodnotu. V jeho hlavičce nejsou povoleny funkční modifikátory (jako `static` či `virtual`) a jakékoliv přístupové modifikátory jsou ignorovány. Přestože v jazyce C++/CLI pracujeme pouze s finalizérem, v kódu MSIL můžeme identifikovat tři metody, jež jsou vygenerovány jako odezva na přítomnost finalizéru v těle naší třídy. Jedná se o následující metody:

- `!D : void()`,
- `Dispose : void(bool)`,
- `Finalize : void()`.

Je-li objekt označen jako nedosažitelný z programového kódu, má automatický správce paměti všechny důvody k tomu, aby takovýto objekt z paměti uvolnil. To, zda dotyčný objekt disponuje svým destruktorem, v tuto chvíli nehraje roli, neboť v jazyce C++/CLI je destruktor interně reprezentován metodou `Dispose`. Automatický správce paměti však nevolá metodu `Dispose`, protože v tomto případě se již jedná o nedeterministickou finalizaci, která musí být provedena pomocí finalizační metody `Finalize`. Metoda `Finalize` není kompilátorem jazyka C++/CLI implicitně generována při definici destruktoru v těle třídy, což však naštěstí není žádná katastrofa. Tato metoda se totiž vytvoří právě tehdy, jestliže do těla řízené třídy vložíme definici finalizéru. Vedle toho kompilátor emituje také instrukce pro sestrojení parametrické metody `Dispose`, která očekává argument v podobě logické hodnoty typu `bool`.

Když je objekt finalizován, je automatickým správcem paměti volána jeho finalizační metoda `Finalize`. Ovšem pozor – tato metoda nepředstavuje finalizér (`!D`). Finalizační metoda aktivuje metodu `Dispose` a odevzdává ji hodnotu `false`, na což metoda `Dispose` zavolá finalizér (`!D`). Finalizér provede příkazy, které jsme zapsali do jeho těla a pak vrátí řízení zpátky metodě `Dispose`. Ta pokračuje dál a volá finalizační metodu primární bázové třídy `System::Object`.

Pokud odkazová třída napsaná v jazyce C++/CLI obsahuje pouze definici finalizéru, kompilátor generuje varování C4461. Tak se nám pokouší naznačit, že vytvoření třídy s finalizérem, ovšem bez destruktoru, neshledává jako dobrý nápad. Proto byste měli pokaždé, když usoudíte, že budete potřebovat finalizér, opatřit třídu rovněž destruktorem. Co se týče dealokačního scénáře, můžete postupovat takhle:

1. Řízené prostředky alokované s objektem třídy uvolněte v destruktoru.

2. Neřízené asociované zdroje zlikvidujte ve finalizéru.

3. Abyste nemuseli duplikovat kód, zavolejte z destruktoru (po uvolnění řízených prostředků) finalizér, který zajistí dealokaci nativních prostředků.

Vlastnosti

Syntaktickou implementaci objektových vlastností v jazyce C++ s Managed Extensions jsme rozebrali v první části této vývojářské příručky. V dřívějším prostředí byly vlastnosti reprezentovány dvojicí členských funkcí třídy, které působily jako přístupové metody k soukromým datovým položkám. Programátoři v C++ s Managed Extensions mohli definovat pro každou vlastnosti dvě metody, `get_` a `set_`, jejichž pomocí bylo možné číst, respektive upravovat hodnoty „ukrytých" datových položek. Přestože kód pracoval spolehlivě, syntaktické řešení nebylo dvakrát oku lahodící. Navíc měli mnozí vývojáři pocit, že kupříkladu v jazyce C# se s vlastnostmi pracuje daleko pohodlněji. Tvůrci C++/CLI vyslyšeli prosby programátorů, takže se můžeme seznámit se zbrusu novým ztvárněním vlastností. Definice vlastnosti již není oddělena na dvě samostatné přístupové metody, ale došlo k jejich sloučení „pod jednu střechu". Vlastnost se definuje pomocí příkazu `property`, v novém prostředí již bez nelíbivých dvojitých značek podtržení.

Novou syntaxi pro definici vlastností v jazyce C++/CLI si budeme demonstrovat na třídě `Lektor`, jejíž pomocí dovedeme generovat virtuální školitele.

```
public ref class Lektor
{
private:
   String^ jmeno, ^prijmeni;
```

```
   String^ zamereni;
public:
   Lektor()
   {
      jmeno = "Jan"; prijmeni = "Nový";
      zamereni = "Java, Cobol, Fortran";
   }
   Lektor(String^ Jmeno, String^ Prijmeni, String^ Zamereni)
   {
      jmeno = Jmeno; prijmeni = Prijmeni;
      zamereni = Zamereni;
   }

   property String^ Jmeno
   {
      String^ get() {return jmeno;}
      void set(String^ nove_jmeno) {jmeno = nove_jmeno;}
   }

   property String^ Prijmeni
   {
      String^ get() {return prijmeni;}
      void set(String^ nove_prijmeni) {prijmeni = nove_prijmeni;}
   }

   property String^ Zamereni
   {
      String^ get() {return zamereni;}
      void set(String^ nove_zamereni) {zamereni = nove_zamereni;}
   }
};
```

Třída `Lektor` není nijak složitá. Osoba lektora je charakterizována pomocí jména, příjmení a zaměření, což je textový popisek profesionálního působení školitele. Tyto tři znaky lektora reflektují soukromé datové položky třídy `Lektor`. V těle třídy můžeme dále najít přetížený instanční konstruktor, jehož první verze inicializuje založenou instanci třídy podle implicitních dat, zatímco druhá verze se na inicializační údaje zeptá programátora. Středem našeho zájmu se však stanou tři vlastnosti s názvy `Jmeno`, `Prijmeni` a `Zamereni`. Všechny tři vlastnosti jsou určeny jak ke čtení, tak i k zápisu hodnot do privátní části instance naší třídy. Modifikovanou syntaxi si nyní blíže představíme u vlastnosti `Jmeno`.

Abychom se nemuseli vracet zpátky, zde je její zdrojový kód ještě jednou:

```
// Definice vlastnosti Jmeno v jazyce C++/CLI.
property String^ Jmeno
{
   String^ get() {return jmeno;}
   void set(String^ nove_jmeno) {jmeno = nove_jmeno;}
}
```

V hlavičce vlastnosti se nachází příkaz `property`, za nímž následuje datový typ vlastnosti. Tento typ určuje, s jakým typem dat budou operovat přístupové metody vlastností. Za hlavičkou vlastnosti přichází její tělo, které je podobně jako tělo funkce, nebo programový blok uzavřeno do složených závorek. V těle vlastnosti jsou situovány metody `get` a `set`, které obhospodařují čtení a ukládání hodnot příslušné datové položky. Jistě nebudete namítat, když prohlásíme, že upravený syntaktický vzhled je velice podobný kódu jazyka C#. Tak schválně: kdybychom přepsali kód vlastnosti `Jmeno` do céčka s mřížkou, skončili bychom takhle:

```
// Definice vlastnosti Jmeno v jazyce C#.
public string Jmeno
{
   get {return jmeno;}
   set {jmeno = value;}
}
```

Přiznáváme, že funkčně ekvivalentní kód v C++/CLI je o něco delší, což je způsobeno explicitním uváděním návratových hodnot obou přístupových metod a také absencí speciální hodnoty `value` jazyka C#, která slouží k inicializaci datové položky v metodě `set`. Nicméně jinak se kód definující vlastnosti obou jazyků vůbec neliší.

Další vlastnosti `Prijmeni` a `Zamereni` využívají totožná syntaktická pravidla. Jsou-li vlastnosti připraveny, můžeme založit testovací instanci třídy a pozměnit hodnoty její datových položek právě pomocí vlastností.

```
Lektor^ lektor1 = gcnew Lektor();

Console::WriteLine("Jméno a příjmení lektora: " + lektor1->Jmeno +
   " " + lektor1->Prijmeni);
Console::WriteLine("Zaměření: " + lektor1->Zamereni);

lektor1->Jmeno = "Petr";
lektor1->Prijmeni = "Burian";
lektor1->Zamereni = "C++, C, C# a Visual Basic";

Console::WriteLine("\nA nyní po změně vlastností...\n");
Console::WriteLine("Jméno a příjmení lektora: " + lektor1->Jmeno +
   " " + lektor1->Prijmeni);
Console::WriteLine("Zaměření: " + lektor1->Zamereni);
Console::Read();
```

Implicitní veřejná jednoduchá dědičnost

Všechny programovací jazyky, které působí na platformě Microsoft .NET Framework 2.0 podporují pouze veřejnou jednoduchou dědičnost. Na tomto faktu se nic nemění, což je ostatně jenom dobře. Faktem však je, že jazyk C++ s Managed Extensions kráčel ve šlépějích svého nativního protějšku, přičemž jako výchozí aplikoval soukromou dědičnost. Jenomže tato forma dědičnosti není v řízeném prostředí povolena. Bez explicitního určení modifikátoru `public` by dědění skončilo chybovým hlášením.

```
// Tato třída nebude kompilátorem jazyka C++ s Managed Extensions
// přeložena. Je to proto, že implicitní soukromé dědění není v tomto
// jazyce podporované.
public __gc class NoveTlacitko : Button
{

};
```

Pro potlačení této výchozí derivace bylo nutné před název bázové třídy, z níž jste odvozovali podtřídu, vložit klíčové slovo `public`.

```
// Nyní je vše v pořádku: podtřída vzniká v procesu jednoduché veřejné
// dědičnosti.
public __gc class NoveTlacitko : public Button
{

};
```

Jestliže vás neustálé vypisování veřejného přístupového modifikátoru `public` obtěžovalo, jazyk C++/CLI má pro vás lék, jehož název zní implicitní veřejné dědění.

```
// Nově lze aplikovat veřejné dědění implicitně.
public ref class NoveTlacitko : Button
{

};
```

Přestože je vám implicitní veřejné dědění k službám, máte-li chuť, můžete i nadále před název bázové třídy vložit klíčové slovo `public`. Kompilátor tento počin nijak nesankcionuje.

Abstraktní třídy

Abstraktní třídy jsou třídy, které nemohou vystupovat jako „továrny na objekty". Řečeno jinak, programátor nemůže vytvořit instanci abstraktní třídy. Místo toho musí napsat další třídu, která bude od abstraktní třídy odvozena (tu již pak samozřejmě instanciovat lze). V jazyce C++ s Managed Extensions se abstraktní třída definuje pomocí klíčového slova `__abstract`, které je uváděno před názvem třídy. V těle abstraktní třídy se mohou vyskytovat jak čisté virtuální funkce, tak i funkce plně definované (v tomto směru není abstraktní třída nijak omezena). Pro názornou ukázku jsme v jazyce C++ s Managed Extensions připravili následující abstraktní třídu:

```
// Definice abstraktní třídy v C++ s Managed Extensions.
public __abstract __gc class Video
{
public:
   void virtual PrehratVideosoubor(String __gc* Cesta) = 0;
};
```

V C++/CLI se modifikátor `__abstract` mění na `abstract` a stejně tak dochází k přesunu modifikátoru – ten se nyní nachází až za jménem abstraktní třídy. Třídu `Video` bychom v jazyce C++/CLI přepsali takto:

```
// Definice abstraktní třídy v C++/CLI.
public ref class Video abstract
{
public:
   void virtual PrehratVideosoubor(String^ Cesta) = 0;};
```

Protože abstraktní třídu nemůžeme využít pro zakládání objektů, zhotovili jsme další třídu s názvem `Videoprehravac`, která bude od naší abstraktní třídy odvozena. V nově definované třídě pak explicitně překryjeme čistou virtuální metodu abstraktní třídy.

```
public ref class Videoprehravac : Video
{
public:
   virtual void PrehratVideosoubor(String^ Cesta) override
   {
      Microsoft::DirectX::AudioVideoPlayback::Video^ video = gcnew
         Microsoft::DirectX::AudioVideoPlayback::Video(Cesta);
      video->Play();
   }
};
```

Všimněte si prosím klíčového slova `override` na samém konci hlavičky virtuální funkce `PrehratVideosoubor`. Programujeme-li v C++/CLI, je zapotřebí virtuální metodu bázové abstraktní třídy výslovně překrýt modifikátorem `override`. Nelze tedy uplatnit implicitní překrývání známé kupříkladu z jazyka C++ s Managed Extensions.

Metoda `PrehratVideosoubor` odkazové třídy `Videoprehravac` uskutečňuje zrození instance třídy `Video` z jmenného prostoru `Microsoft::DirectX::AudioVideoPlayback`. Přetíženému instančnímu konstruktoru třídy `Video` je předána plně kvalifikovaná cesta k souboru s multimediálním obsahem. Data ze specifikovaného souboru jsou poté přečtena a na obrazovce počítače zobrazena pomocí metody `Play`.

UPOZORNĚNÍ

Aby uvedený programový kód pracoval jako hodinky, je nutno vložit do projektu Visual C++ 2005 odkaz na sestavení Microsoft.DirectX.AudioVideoPlayback.dll. Toto sestavení je součástí řízeného rozhraní technologie Microsoft DirectX 9. Tím chceme říci, že plynulý běh ukázky požaduje instalaci DirectX 9 na vašem počítači (SDK pro DirectX 9 můžete získat na webových stránkách společnosti Microsoft). Sestavení přidáte do vašeho projektu Visual C++ 2005 takto:

1. Otevřete nabídku **Project** a klepněte na poslední položku **Properties**.
2. Jakmile uvidíte dialogové okno **Property Pages**, zaměřte se na stromovou strukturu nacházející se na levé straně.
3. Otevřete uzel **Common Properties**, případně také položku **References**.
4. Stiskněte tlačítko **Add New Reference...**.
5. Spatříte-li dialog **Add Reference**, ujistěte se, že je vybrána záložka **.NET**.
6. Vyhledejte sestavení Microsoft.DirectX.AudioVideoPlayback a aktivujte tlačítko **OK**. Nyní je reference na sestavení přidána do projektu, což znamená, že můžete pracovat s datovými typy v tomto sestavení definovanými.

Mimochodem, sestavení Microsoft.DirectX.AudioVideoPlayback je nainstalované v globální mezipaměti sestavení (Global Assembly Cache, GAC).

Další výpis zdrojového kódu ukazuje, že přehrání videosouboru je otázkou dvou řádků programového kódu:

```
Videoprehravac^ obj = gcnew Videoprehravac();
obj->PrehratVideosoubor("d:\\video.wmv");
Console::Read();
```

UPOZORNĚNÍ	
	Společné běhové prostředí CLR vývojově-exekuční platformy .NET Framework 2.0 generuje při použití sestavení Microsoft.DirectX.AudioVideoPlayback.dll výjimku LoaderLock. Přesněji řečeno, tato výjimka je zachycena řízeným ladícím asistentem. Pro zabezpečení bezproblémového běhu programu musíte rozsah působnosti ladícího asistenta omezit. To uděláte následovně: 1. Otevřete nabídku **Debug** a klepněte na příkaz **Exceptions...**. 2. V okně **Exceptions** rozviňte kořenový uzel **Managed Debugging Assistants**. 3. Najděte položku **LoaderLock** a zrušte u ní zatržení v sloupci **Thrown**. 4. Spusťte aplikaci a vyzkoušejte, zda pracuje tak, jak má.

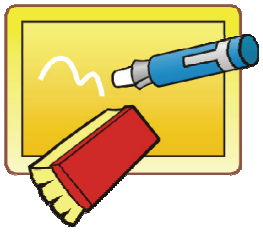
Zapečetěné třídy

Usoudí-li vývojář, že napsaná třída disponuje plně definovanou funkcionalitou, přičemž není předpoklad, že by třída mohla v budoucnosti sloužit pro odvozování dalších tříd, může být opatřena klíčovým slovem `sealed`. Modifikátor `sealed` jazyka C++/CLI je nástupcem klíčového slova `__sealed`, které znáte z C++ s Managed Extensions. Dříve stálo klíčové slovo `__sealed` před názvem třídy, nyní se jeho pozice posouvá za název třídy. Následující řádky znázorňují, jak se zapečetěná třída definovala v jazyce C++ s Managed Extensions a jaké změny pro ni platí při přechodu do C++/CLI. Jenom doplňme, že třída slouží pro automatizaci aplikace Microsoft Office Outlook 2003.

Definice zapečetěné třídy v C++ s Managed Extensions:

```
// Import jmenného prostoru.
using namespace Microsoft::Office::Interop;

// Klíčové slovo __sealed mění standardní třídu na třídu zapečetěnou.
public __sealed __gc class AutoOutlook
{
private:
   Outlook::ApplicationClass __gc* AppOutlook;
   Outlook::NameSpace __gc * JmennyProstor;
   Outlook::MAPIFolder __gc * Slozka;
public:
   void Spustit()
   {
      // Zrození nové instance třídy ApplicationClass, která představuje
      // novou relaci aplikace Outlook 2003.
      AppOutlook = __gc new Outlook::ApplicationClass();
      // Výběr cílového prostoru jmen.
      JmennyProstor = AppOutlook->GetNamespace(S"MAPI");
      // Zde nařizujeme, aby byla zobrazena složka s kalendářem.
      Slozka = JmennyProstor->GetDefaultFolder
        (Outlook::OlDefaultFolders::olFolderCalendar);
      // Nakonec nakonfigurovanou aplikaci Outlook 2003 zviditelňujeme.
      Slozka->Display();
   }
};
```

POZNÁMKA

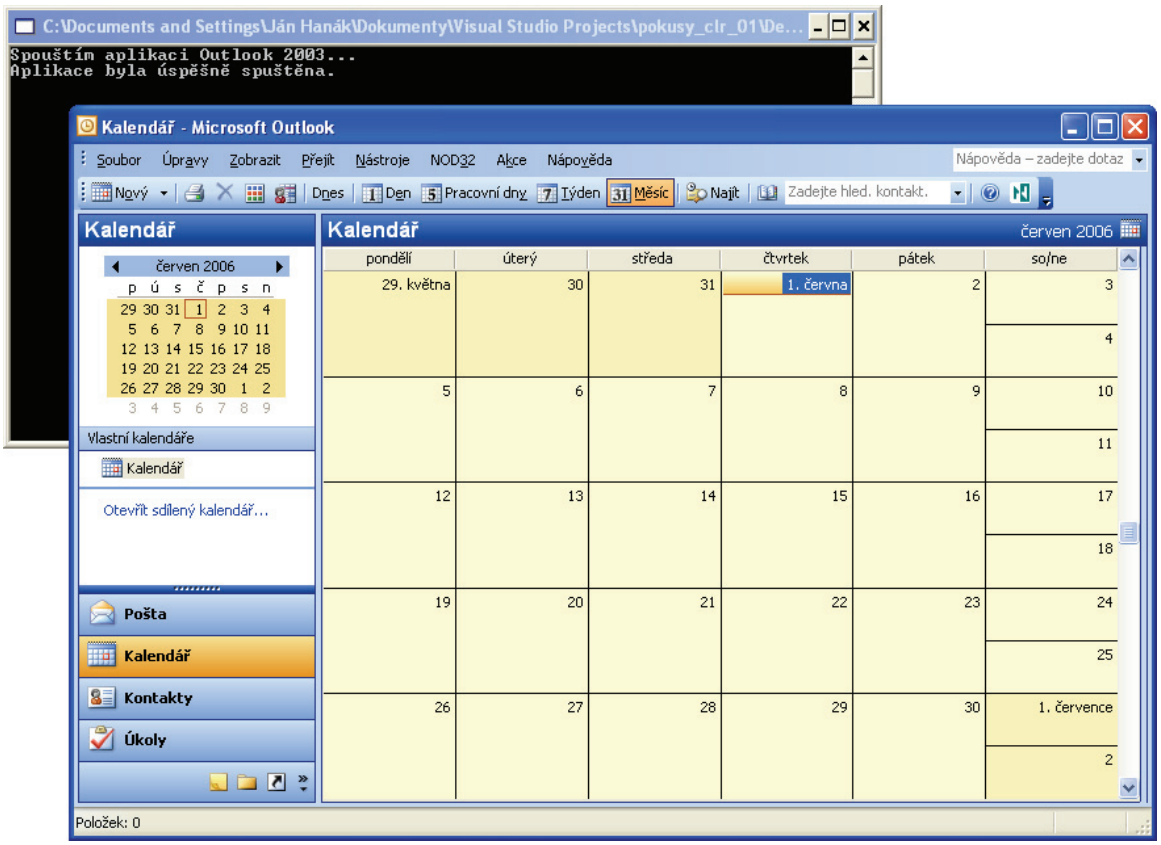


Pro zdárné přeložení kódu zapečetěné třídy musíte do svého aplikačního projektu importovat referenci na primární sestavení vzájemné spolupráce (Primary Interop Assembly, PIA) pro Microsoft Office Outlook 2003. Toto sestavení PIA obsahuje řízená metadata a definice řízených datových typů, které odpovídají nativním datovým typům uloženým v objektových knihovnách Office. Ačkoliv se při automatizaci aplikace Outlook 2003 používá právě příslušné sestavení PIA, do integrovaného vývojového prostředí Visual C++ .NET 2003 musíte vložit odkaz na objektovou knihovnu Microsoft Outlook 11.0 Object Library.

Použití třídy v C++ s Managed Extensions:

```
int _tmain()
{
   Console::WriteLine(S"Spouštím aplikaci Outlook 2003...");
   AutoOutlook __gc * obj = __gc new AutoOutlook();
   obj->Spustit();
   Console::WriteLine(S"Aplikace byla úspěšně spuštěna.");
   Console::Read();
   return 0;
}
```

Výsledkem práce instance zapečetěné třídy je spuštění aplikace Outlook 2003 a zobrazení složky s kalendářem. V příkazovém řádku konzolové aplikace se přitom zobrazují zprávy o spouštění aplikace.



Obr. 2.6: Automatizace aplikace Microsoft Office Outlook 2003

Nyní se přenesme do světa C++/CLI. Přepíšeme-li definici zapečetěné třídy do tohoto programovacího jazyka, dostaneme níže uvedený fragment zdrojového kódu:

```
using namespace Microsoft::Office::Interop;

public ref struct AutoOutlook sealed
{
private:
   ::Outlook::ApplicationClass^ AppOutlook;
   ::Outlook::NameSpace^ JmennyProstor;
   ::Outlook::MAPIFolder^ Slozka;
public:
   void Spustit()
   {
      AppOutlook = gcnew ::Outlook::ApplicationClass();
      JmennyProstor = AppOutlook->GetNamespace("MAPI");
      Slozka = JmennyProstor->GetDefaultFolder
         (::Outlook::OlDefaultFolders::olFolderCalendar);
      Slozka->Display();
   }
};
```

Založení instance zapečetěné třídy je pak již velice snadné:

```
int main(array<System::String ^> ^args)
{
   Console::WriteLine("Spouštím aplikaci Outlook 2003...");
   AutoOutlook^ obj = gcnew AutoOutlook();
   obj->Spustit();
   Console::WriteLine("Aplikace byla úspěšně spuštěna.");
   Console::Read();
}
```

I když je možné zakládat instance zapečetěné třídy, není povoleno tuto třídu použít jako bázi pro odvozování podtříd. Kdybychom se tudíž pokusili provést něco takovéhoto

```
// Pozor! Abstraktní třída nesmí vystupovat v roli třídy bázové.
public ref class AutoOutlook2 : AutoOutlook
{

};
```

kompilátor by nás co nevidět zastavil s chybovým hlášením.

Závěr

Vážení příznivci jazyka C++,

právě jste dočetli poslední stránku vývojářské příručky **Programujeme v jazycích C++ s Managed Extensions a C++/CLI**. Pokud se knize podařilo vzbudit váš zájem o vývoj aplikací .NET v řízeném C++, pak splnila svůj cíl. Když jsme publikaci koncipovali, chtěli jsme podat srozumitelné informace dvěma skupinám programátorům: jednak těm, kteří vlastní produkt Visual C++ .NET 2003 a také těm, kteří by rádi přešli na verzi Visual C++ 2005. Zatímco v první části jsme rozebrali základy programování v jazyce C++ s Managed Extensions, v druhé jsme se věnovali výkladu stěžejních témat, s nimiž se musejí vývojáři obeznámit v případě, že hodlají zabrousit do tajů jazyka C++/CLI.

Jazyky C++ s Managed Extensions a C++/CLI nepatří zrovna k triviálním programovacím jazykům. Dokonce bychom mohli tvrdit, že z trojice Visual Basic, C# a řízené C++ je právě poslední adept tím nejsložitějším a snad také nejtěžším k naučení. To je patrně jediná nevýhoda řízeného C++. Visual Basic a C# totiž mnoho věcí před programátory ukrývají, takže programování aplikací je v těchto prostředích přece jenom snazší. Na druhou stranu, jazyky C++ s Managed Extensions a C++/CLI dále rozvíjejí úspěšnou tradici nativního „céčka s dvěma plusy", přičemž vám poskytují vysoce sofistikovaný programovací aparát, s nímž můžete kontrolovat i ty nejjemnější aspekty programování. Řízené C++ je rovněž velice dobrou volbou do budoucna, neboť ve své poslední verzi známé jako C++/CLI se již jedná o promyšlený a propracovaný nástroj na tvorbu softwarových aplikací.

Přestože jsme se snažili tuto vývojářskou příručku naplnit co možná nejzajímavějšími tématy, jistě uznáte, že při tak košatém jazyce, jakým řízené C++ bezesporu je, jsme se ani zdaleka nemohli dotknout všech vzrušujících programátorských zákoutí. To však konec konců ani nebylo naším záměrem: spíše jsme se snažili přiblížit vám jazyky C++ s Managed Extensions a C++/CLI a naučit vás nezbytné programátorské základy. Budete-li mít chuť, můžete ve studiu dále pokračovat. Vynaložené úsilí a energie se vám zcela jistě vrátí, ať už v podobě vyšší úrovně znalostí či rozšíření vašich vývojářských obzorů. Každopádně vám na této cestě přejeme hodně štěstí a pracovního entusiasmu.

Autor a společnost Microsoft

Informace o autorovi

Ing. Ján Hanák vystudoval Obchodní fakultu Ekonomické univerzity v Bratislavě. Působí jako programátor, vývojář, lektor, technický konzultant a spisovatel, přičemž se specializuje na vývoj širokého spektra počítačových aplikací. Při své práci využívá především tyto programovací jazyky: Visual Basic, VBA, C#, C++ s Managed Extensions, C++/CLI, nativní C++ a C.

V roce 2006 napsal publikaci **C# - Praktické příklady**, kterou vydalo nakladatelství Grada Publishing. Mezi jeho další významná díla patří: **VB 6.0 → VB 2005: Přecházíme z jazyka Visual Basic 6.0 na jazyk Visual Basic 2005** (Microsoft ČR, 2005), **Visual Basic .NET 2003 – Začínáme programovat** (Grada Publishing, 2004) a **Přecházíme na platformu Microsoft .NET** (překlad, Microsoft ČR, 2004).

Svá důvtipná programátorská řešení publikoval v počítačových časopisech CHIP, PC REVUE a INFOWARE, kde také vedl několik seriálů a rubrik pro programátory, vývojáře a softwarové architekty. Rovněž aktivně spolupracuje s magazíny PC World a Computerworld.

V současnosti působí na Katedře aplikované informatiky Fakulty hospodářské informatiky Ekonomické univerzity v Bratislavě. Své znalosti rád předává dalším IT odborníkům, kupříkladu také na technických seminářích realizovaných společností Microsoft.

V roce 2006 společnost Microsoft ocenila jeho technické znalosti a pozitivní vliv na vývojářskou komunitu a odměnila jej titulem **Microsoft Most Valuable Professional (MVP) s kompetencí Visual Developer – Visual C++**.

Pokud chcete autorovi něco sdělit, můžete jej zastihnout na následující adrese elektronické pošty: hanja@stonline.sk. Za vaše reakce vám bude velice vděčný.