Možnosti využití unmanaged kódu v jazyce C#

Bc. Pavel Kharytonchyk

Diplomová práce
2019

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky
akademický rok: 2018/2019

ZADÁNÍ DIPLOMOVÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení:	**Bc. Pavel Kharytonchyk**
Osobní číslo:	**A16225**
Studijní program:	**N3902 Inženýrská informatika**
Studijní obor:	**Informační technologie**
Forma studia:	**kombinovaná**
Téma práce:	**Možnosti využití unmanaged kódu v jazyce C#**
Téma anglicky:	**The Use of Unmanaged Code in C# Language**

Zásady pro vypracování:

1. **Popište možnosti řízení pamětí v prostředí Common Language Runtime a způsoby využiti unmanaged kódu v jazyce C#.**
2. **Rozeberte vybrané datové struktury z hlediska optimalizace pomocí unmanaged kódu.**
3. **Navrhněte vhodné řešení z hlediska přístupu k datovým strukturám v paměti RAM.**
4. **Vytvořte ukázkové kódy představující klíčové prvky řešení.**
5. **Demonstrujte výsledky a formulujte závěr.**

Rozsah diplomové práce:

Rozsah příloh:

Forma zpracování diplomové práce: **tištěná/elektronická**

Seznam odborné literatury:

1. **GOLDSHTEIN, Sasha, Dima ZURBALEV a Ido FLATOW. Pro .NET performance. Berkeley, Calif.: Apress, [2012]. Expert's voice in .NET. ISBN 978-1-4302-4458-5.**
2. **RICHTER, Jeffrey. CLR via C#. Fourth edition. Redmond, Washington: Microsoft, [2012]. ISBN 978-0-735-66745-7.**
3. **WIRTH, Niklaus. Algorithms data structures=programs. Englewood Cliffs, N.J.: Prentice-Hall, c1976. ISBN 01-302-2418-9.**
4. **AHO, Alfred V, John E HOPCROFT a Jeffrey D ULLMAN. Data structures and algorithms. Reading, Mass.: Addison-Wesley, c1983. ISBN 02-010-0023-7.**
5. **NAGEL, Christian. Professional C# 6 and .Net Core 1.0. Indianapolis, IN: John Wiley, 2016. ISBN 9781119096603.**
6. **Standard ECMA-372 C++/CLI Language Specification [online]. [cit. 2018-09-24]. Dostupné z: http://www.ecma-international.org/publications/standards/Ecma-372.htm**

Vedoucí diplomové práce: **Ing. Erik Král, Ph.D.**
Ústav počítačových a komunikačních systémů

Datum zadání diplomové práce: **3. prosince 2018**

Termín odevzdání diplomové práce: **15. května 2019**

Ve Zlíně dne 7. prosince 2018

doc. Mgr. Milan Adámek, Ph.D.
děkan

prof. Mgr. Roman Jašek, Ph.D.
garant oboru

Prohlašuji, že

- beru na vědomí, že odevzdáním diplomové práce souhlasím se zveřejněním své práce podle zákona č. 111/1998 Sb. o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších právních předpisů, bez ohledu na výsledek obhajoby;
- beru na vědomí, že diplomová práce bude uložena v elektronické podobě v univerzitním informačním systému dostupná k prezenčnímu nahlédnutí, že jeden výtisk diplomové/bakalářské práce bude uložen v příruční knihovně Fakulty aplikované informatiky Univerzity Tomáše Bati ve Zlíně a jeden výtisk bude uložen u vedoucího práce;
- byl/a jsem seznámen/a s tím, že na moji diplomovou práci se plně vztahuje zákon č. 121/2000 Sb. o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon) ve znění pozdějších právních předpisů, zejm. § 35 odst. 3;
- beru na vědomí, že podle § 60 odst. 1 autorského zákona má UTB ve Zlíně právo na uzavření licenční smlouvy o užití školního díla v rozsahu § 12 odst. 4 autorského zákona;
- beru na vědomí, že podle § 60 odst. 2 a 3 autorského zákona mohu užít své dílo – diplomovou práci nebo poskytnout licenci k jejímu využití jen připouští-li tak licenční smlouva uzavřená mezi mnou a Univerzitou Tomáše Bati ve Zlíně s tím, že vyrovnání případného přiměřeného příspěvku na úhradu nákladů, které byly Univerzitou Tomáše Bati ve Zlíně na vytvoření díla vynaloženy (až do jejich skutečné výše) bude rovněž předmětem této licenční smlouvy;
- beru na vědomí, že pokud bylo k vypracování diplomové práce využito softwaru poskytnutého Univerzitou Tomáše Bati ve Zlíně nebo jinými subjekty pouze ke studijním a výzkumným účelům (tedy pouze k nekomerčnímu využití), nelze výsledky diplomové/bakalářské práce využít ke komerčním účelům;
- beru na vědomí, že pokud je výstupem diplomové práce jakýkoliv softwarový produkt, považují se za součást práce rovněž i zdrojové kódy, popř. soubory, ze kterých se projekt skládá. Neodevzdání této součásti může být důvodem k neobhájení práce.

Prohlašuji,

- že jsem na diplomové práci pracoval samostatně a použitou literaturu jsem citoval, v případě publikace výsledků budu uveden jako spoluautor;
- že odevzdaná verze diplomové práce a verze elektronická nahraná do IS/STAG jsou totožné.

Ve Zlíně dne 10. května 2019

Pavel Kharytonchyk, v.r.

……………………………….

podpis diplomanta

ABSTRAKT

Diplomová práce zkoumá a popisuje možnosti využití unmanaged kódu pomocí programovacího jazyka C# v řízeném prostředí Common Language Runtime (CLR). Práce se skládá celkem z pěti částí. Teoretická část se převážně věnuje rozboru toho, jak CLR pracuje s pamětí. Dále jsou krátce rozebrány způsoby interoperability mezi řízeným a neřízeným kódem a stručně je popsána datová struktura, která byla využita při testování. V praktické části je za použití jazyků C#, C a C++ a technologií Platform Invoke a C++/CLI realizovaná komunikace mezi řízeným a nativním kódem. Dále jsou popsány nejužitečnější části jednotlivých implementací. Následuje grafická prezentace a porovnání testů navržených implementací. V závěru práce jsou vyhodnoceny výsledky testů a je uvedeno doporučení pro správné používání interoperability v .NET.

Klíčová slova: C++/CLI, Common Language Runtime, nativní kód, Platform Invoke, .NET Framework, programovací jazyky C#, C, C++, unmanaged kód.

ABSTRACT

This diploma thesis explores and describes the ways of using unmanaged code by the C# programming language in the managed Common Language Runtime (CLR) environment. The theoretical part is mainly devoted how CLR works with memory. Further, the methods of interoperability between managed and native code are briefly discussed. Farther the data structure that using in tests is briefly described. The second part is practical, in which is implemented the communication between managed and native code, using C#, C and C++ with Platform Invoke and C ++/CLI technologies. Then author describes the most useful parts of each implementations and graphically presents results of his own tests and their comparisons. At the end of this thesis, the author evaluates the most impotent test results and give his recommendations for the correct use of interoperability in .NET.

Keywords: C++/CLI, Common Language Runtime, native code, Platform Invoke, .NET Framework, programming languages C#, C, C++, unmanaged code.

Rád bych na tomto místě poděkoval svému vedoucímu diplomové práce Ing. et Ing. Eriku Královi, Ph.D. za vstřícný přístup, všechny užitečné rady a také čas, který mi při vedení této práce věnoval. Rovněž bych chtěl poděkovat své rodině a přátelům, kteří mě při psaní podporovali.

Prohlašuji, že odevzdaná verze diplomové práce a verze elektronická nahraná do IS/STAG jsou totožné.

OBSAH

ÚVOD **9**
I TEORETICKÁ ČÁST **10**
1 CLR **11**
1.1 COMMON TYPE SYSTEM 12
1.2 MANAGED A UNMANAGED KÓD 13
1.3 MOŽNOSTI ŘÍZENÍ PAMĚTI V PROSTŘEDÍ CLR 13
1.3.1 Managed heap 13
1.3.2 Alokace zdrojů 14
1.3.3 Garbage collector 16
1.3.3.1 Generace objektů 18
1.3.4 Velké objekty 21
1.3.5 Marshalling 22
1.3.6 Uvolnění zdrojů pomocí mechanizmu finalizace 23
2 ZPŮSOBY VYUŽITI UNMANAGED KÓDU V JAZYCE C# **25**
2.1 PLATFORM INVOKE 25
2.2 C++/CLI 26
2.2.1.1 C++ Implicit Interop 27
2.3 KLÍČOVÉ SLOVO „UNSAFE“ 27
3 DATA **29**
3.1 STROMOVÉ STRUKTURY 29
3.2 AVL B-TREE 31
II PRAKTICKÁ ČÁST **32**
4 IMPLEMENTACE **33**
4.1 BINÁRNÍ STROM 33
4.2 INTEROP 33
4.2.1 P/Invoke 33
4.2.1.1 P/Invoke Interop Asistent 34
4.2.1.2 C DLL 34
4.2.1.3 C++ Dll 36
4.2.2 C++/CLI 37
4.2.2.1 C++/CLI 37
4.2.2.2 C++/CLI mixed mode 39
4.2.3 C# managed 41
5 BENCHMARKING **42**
5.1 SBĚR DAT 42
5.2 METODIKA 42
5.3 IMPLEMENTACE 43
5.3.1 Logování dat 43
5.4 PREZENTACE VÝSLEDKŮ 44
5.4.1 Grafy 45
5.5 VYHODNOCENÍ VÝSLEDKŮ 49
5.5.1 Používání nativního kódu v managed prostředí 50

ZÁVĚR 51
SEZNAM POUŽITÉ LITERATURY 52
SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK 53
SEZNAM OBRÁZKŮ 55
SEZNAM GRAFŮ 56
SEZNAM TABULEK 57
SEZNAM PŘÍLOH 58

ÚVOD

Zvýšení výkonu algoritmů a aplikací je nesmírně důležitým aspektem technologického vývoje. Může poskytnout určitou konkurenční výhodu a zajistit, aby uživatelé měli radost z aplikací s rychlou odezvou.

Budeme-li však uvažovat v kontextu technologie .NET, vysoká produktivita programátorů byla a vždy bude nejsilnější přínosnou hodnotou platformy .NET. Třeba realizace sběru odpadů je nejdůležitějším mechanismem, který ke zvýšení produktivity přispívá nejvíce. Důvodem není jen to, že eliminuje celou řadů chyb při práci s pamětí, ale také fakt, že umožňuje psát knihovny tříd, aniž by je přeplňoval různými dohodami o přidělení prostředků. Například nenutí implementovat dočasnou vyrovnávací paměť nebo definovat pravidla o tom, kdo má paměť uvolňovat. Výrazným pozitivem je i přísná kontrola typu, která umožňuje odhalit záměry programátora a najít mnoho běžných chyb před spuštěním programu. Je třeba ale i připustit, že následkem těchto vítaných vlastností platformy .NET je určité zpomalení, například při práci s řízenou pamětí kvůli kontrolám přesahu hranic pole apod.

Z výše uvedených důvodů byla jako hlavní cíl této práce zvolena analýza možností využití nativního kódu v řízeném prostředí CLR. I když se údržba projektu ve dvou různých jazycích jeví jako velmi problematická, zejména při použití nespravovaného kódu, který je opravdu obtížné implementovat, ladit a udržovat, jde o zajímavé a perspektivní řešení. Možnost naimplementovat funkcionalitu, která bude rychlejší, si zaslouží pozornost zejména v kritických oblastech nebo ve vysoce zatížených aplikacích. Dalším důvodem pro interoperabilitu může být již existující implementace potřebné funkcionality v unmanaged podobě, neboť není třeba přepisovat již hotové řešení od začátku, ale použít toto řešení přímo v platformě .NET.

I. TEORETICKÁ ČÁST

1 CLR

Michaelis [1] uvádí, že **CLR** (*Common Language Runtime*) je specifická implementace standardu CLI (*Common Language Infrastructure*) pro .NET Framework nebo jak píše Richter [2] je to mnohajazykové běhové prostředí, které zabezpečuje integraci mezi formální jazyky, které jsou do CRL kompilovatelné. Díky standardní sadě typů a metadat dovoluje objekty vytvořené v určitém jazyce použít v jiném [2].

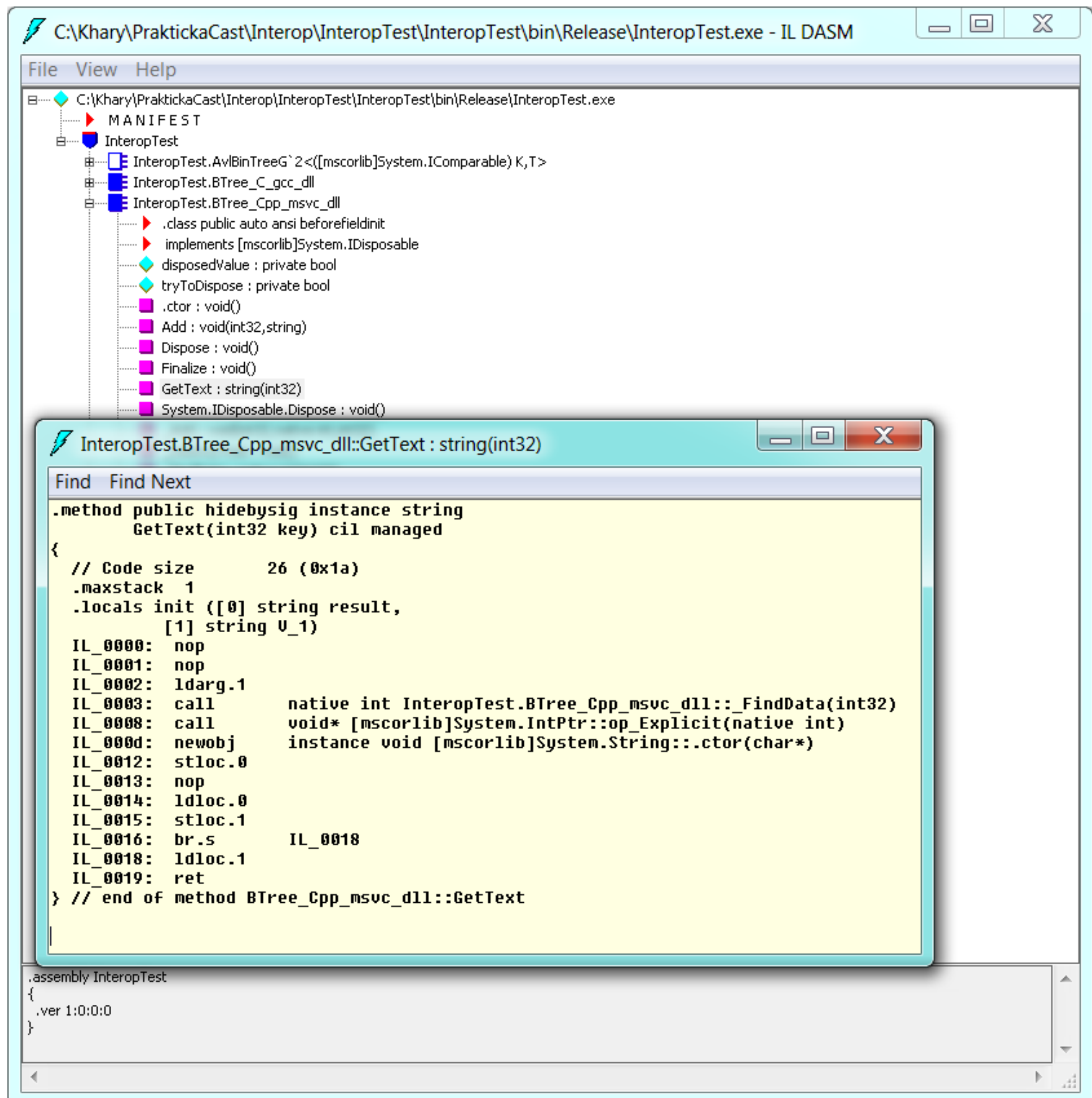
Jinými slovy CLR je mechanismus, který dovoluje aplikaci volat funkce ve správném pořadí a tím ovlivňovat data. Veškeré klíčové vlastnosti CLR, jako je správa paměti, načtení builds (sestavení), bezpečnost, zpracování výjimek nebo synchronizace, jsou k dispozici v libovolném jazyce programování používajícím toto prostředí. Například při zpracování chyb se běhové prostředí opírá o výjimky a tím pádem veškeré programovací jazyky používající CLR posílají také zprávy o chybách pomocí stejného mechanismu výjimek. CLR například umožňuje vytvářet programová vlákna, to znamená, že libovolný formální jazyk, který je do CLR kompilovatelný, může také vlákna vytvářet [2].

Na obrázku 1 jsou zobrazená metadata projektu *InteropTest* a IL kód metody *GetText*, která pro volaní nativní metody používá technologii *P/Invoke*. Z obrázku je vidět, že, veškerý zdrojový kód (buď C#, nebo jiný) musí být před spuštěním kódu v CLR zkompilován. Kompilace v .NET probíhá ve dvou krocích:

1. kompilace zdrojového kódu do Microsoft Intermediate Language (IL)
2. kompilace IL do strojového kódu pro konkrétní platformu pomocí CLR [2].

IL (Intermediate Language, „mezijazyk“) je kód ve speciálním jazyce připomínající assembler, ale vytvořený pro .NET. Do něj se kompilují zdrojové kódy napsané ve vysokoúrovňových programovacích jazycích, což zaručuje nezávislost na jediném vybraném jazyce [2].

Metadata jsou sadou datových tabulek, které popisují, co je definováno v modulu, například typy a jejich členy [2]. Každý kompilátor do CLR (kromě generování kódu IL) musí pro každý řízený modul vytvořit kompletní metadata. Metadata také obsahují tabulky, které označují, na co se vztahuje spravovaný modul, například importované typy a jejich členy, a jsou vždy spojeny se souborem obsahujícím kód IL. Ve skutečnosti jsou metadata vždy vložena spolu s kódem do stejného souboru EXE nebo DLL, takže jsou od sebe neoddělitelná. Dále díky tomu, že kompilátor metadata a kód generuje společně a následně je spojuje s výsledným řízeným modulem, metadata vždy odpovídají IL kódu, který popisují [2].



Obr. 1. Metadata a IL kód metody GetText.

1.1 Common Type System

Bez ohledu na programovací jazyk bude výsledná aplikace interně používat určité datové typy, a právě proto CLI (a to znamená i CLR) zahrnuje *Common Type System (CTS)*. CTS definuje, jak jsou typy strukturovány a rozloženy v paměti, stejně jako koncepty a chování spojené s typy. Spolu s informacemi o datech uložených v typu zahrnuje direktivy manipulace s typem. Vzhledem k tomu, že účelem standardu je dosažení interoperability mezi jazyky, norma CTS se vztahuje k tomu, jak se typy objevují a chovají na vnější hranici jazyka. To znamená, že CLR jako běhové prostředí musí hledat dodržení stanovených konvencí v CTS [1].

V rámci CTS jsou typy klasifikovány do dvou kategorií:

- Hodnotové typy (*values*) – jsou bitové vzory používané k reprezentaci základních typů, jako jsou celá čísla a znaky nebo i složitější data reprezentovaná strukturami.
- Objekty obsahující označení typu objektu (pomáhá to při kontrole typu) – objekty mají identitu, která činí každou instanci unikátní. Objekty mají navíc sloty, které mohou obsahovat další typy (buď hodnotové, nebo odkazy na objekty). Na rozdíl od hodnotových typů změna obsahu slotu nemění identitu objektu [1].

Tyto dvě kategorie typů se převádějí přímo na syntaxi C#, která poskytuje prostředky pro deklaraci každého typu [1].

1.2 Managed a unmanaged kód

V kontextu .NET a jazyka C# je důležité poznamenat, že tento jazyk lze použít pouze k vytvoření softwaru, který je hostován v rámci runtime .NET. C# se například nikdy nepoužívá k vytvoření nativního serveru COM nebo unmanaged aplikace ve stylu C/C++. Oficiálním termínem používaným k popisu kódu zaměřeného na .NET runtime je *řízený kód* (*managed code*). Binární jednotka, která obsahuje řízený kód, se nazývá *assembly*. Naopak kód, který nemůže být přímo hostován runtime .NET, se nazývá *nespravovaný kód* (*unmanaged* code). Jak již bylo zmíněno (a podrobněji je popsáno v této kapitole a následujícím textu), platformu .NET lze provozovat na různých operačních systémech [3].

1.3 Možnosti řízení paměti v prostředí CLR

1.3.1 Managed heap

Každý program používá různé prostředky, například soubory, vyrovnávací paměť, obrazovku, síťová připojení, databáze atd. V objektově orientovaném prostředí každý typ identifikuje určitý druh zdroje, který je tomuto programu k dispozici, a chceme-li jej použít, pro reprezentaci tohoto typu musí být alokována paměť, která se bere z tzv. *Managed heap* [2].

Při použití *resource (zdroje)* jsou platná následující pravidla:

1. Alokovat paměť pro typ reprezentující zdroj, obvykle se to provádí pomocí operátora *new* v C#.
2. Inicializovat přidělenou paměť převedením zdroje do počátečního stavu např. pomocí konstruktoru.
3. Použít zdroj pomocí volání na členy typu (i opakovaně).
4. V rámci procesu čištění zničit stav zdroje.
5. Uvolnit paměť. Za tuto fázi je zodpovědný pouze garbage collector [2].

Toto zdánlivě jednoduché paradigma se stalo jedním z hlavních zdrojů chyb pro programátory, kteří musí ručně spravovat paměť. Dokud ale je vyvíjeno bezpečně, tzn. bez použití klíčového slova *unsafe* v C#, jakékoli poškození nebo únik paměti je ve standardní situaci nemožný. Dále je takový vývoj zjednodušen tím, že pro většinu typů, které vývojáři pravidelně používají, není zničení stavu zdroje povinný krok (viz krok č. 4). Takto řízená halda poskytuje vývojáři kromě odstranění chyb spojených s ruční správou paměti také jednoduchý programovací model: program přiděluje a inicializuje *resource* a pak jej používá tak dlouho, dokud je ve zdroji potřeba. U většiny typů uvolnění zdrojů není potřeba, paměť totiž bude uvolněna GC [2].

I při použití instancí typů, které vyžadují řízené uvolnění prostředků, zůstává programovací model stále jednoduchý. Avšak občas by uvolnění prostředků mělo proběhnout co nejdříve a bez čekání na zásah GC. V takovém případě lze v těchto třídách zavolat metodu *Dispose*, která slouží pro uvolnění zdrojů podle vlastního rozvrhu. Typy, které vyžadují čištění, zpravidla využívají systémové prostředky nízké úrovně – soubory, sokety nebo databázová připojení [2].

1.3.2 Alokace zdrojů

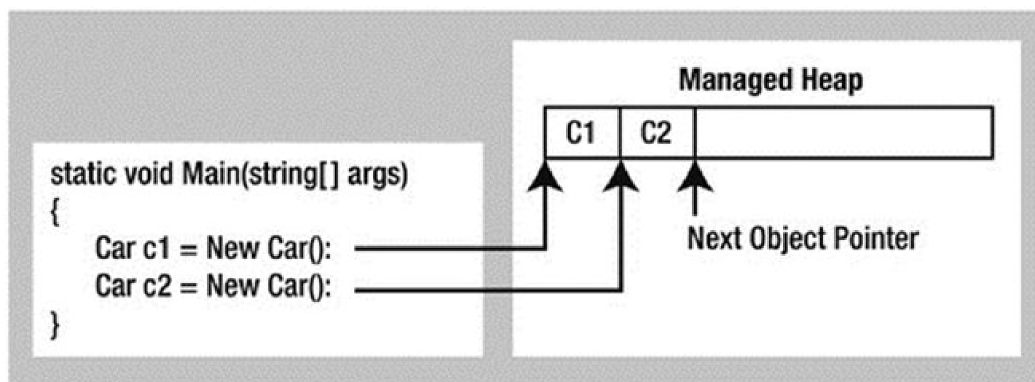
V CLR se paměť pro všechny prostředky přiděluje z *Managed heap*. Při inicializaci procesu CLR rezervuje adresní prostor pro řízenou haldu a také vytváří ukazatel, který lze nazvat *NextObjPtr*. Tento ukazatel určuje místo na haldě, kde bude alokována paměť pro následující objekt, a zpočátku odkazuje na základní adresu této alokované oblasti adresního prostoru [2].

Podle zaplnění oblasti objekty CLR automatický rozšiřuje alokovanou oblast až do zaplnění celého adresního prostoru. Objem paměti pro aplikaci je tak omezen virtuálním adresovým

prostorem procesu, tzn. že pro 32bitové procesy může byt přiděleno do 1,5 gigabajtů paměti a přibližně 8 terabajtů pro 64bitové procesy [2].

Podívejme se, jak běhové prostředí alokuje paměť při použití operátora *new* v C# (Obr. 2):

1. Spočítá počet bajtů potřebných pro umístění všech polí typu (a všech polí zděděných ze základního typu).
2. K získané hodnotě přidá počet bajtů potřebných pro umístění systémových polí objektu. Každý objekt má dvojici takových polí: objekt-typ a index bloku synchronizace. V 32bitových aplikacích vyžaduje každé z těchto polí 32 bitů, což zvyšuje velikost každého objektu o 8 bajtů, a v 64bitových aplikacích každé takové pole vyžaduje 64 bitů, což znamená narůst alokované paměti o 16 bajtů v každém objektu.
3. Zkontroluje, zda je ve vyhrazené oblasti dostatek bajtů k přidělení paměti objektu. Pokud je ve spravované haldě dostatek místa pro objekt, paměť se k němu přidělí, a to počínaje adresou, na kterou se odkazuje ukazatel *NextObjPtr*, dále bajty, které zabíral *NextObjPtr*, jsou vynulované. Potom se volá konstruktor typu, který předává *NextObjPtr* jako parametr *this*, a operátor *new* vrací odkaz na objekt. Před vrácením této adresy se *NextObjPtr* přesune na první adresu za objektem a odkazuje na adresu, ve které bude rozmístěn nový objekt na haldě [2].



Obr. 2. Alokace objektů na Managed heap [3, s. 482].

U *Managed heap* je přidělení paměti pro objekt prováděno jednoduchým povýšením ukazatele a tato operace se provádí téměř okamžitě [2].

V mnoha aplikacích jsou objekty vytvořené téměř ve stejné chvíli navzájem úzce spjaty a také se často používají přibližně ve stejnou dobu, například za objektem *FileStream* se obvykle hned vytváří objekt *BinaryWriter*, aplikace pak přistupuje k objektu *BinaryWriter*, jehož vnitřní kód používá *FileStream*. V prostředí s automatickým sběrem odpadu jsou nové

objekty ukládány do paměti neustále za sebou, což zvyšuje výkon díky blízkému rozmístění na haldě, ale také to znamená, že rozptyl využité paměti v procesu bude menší než u podobné aplikace pracující v neřízeném prostředí. S největší pravděpodobností budou všechny objekty použité v programu umístěny v mezipaměti procesoru. Aplikace bude přistupovat k těmto objektům s fenomenální rychlostí, protože procesor bude provádět většinu svých operací bez chyb známých jako *Cache Miss* (data v mezipaměti nejsou nalezena a musí být přečtena z operační paměti), které zpomalují přístup k RAM [2].

1.3.3 Garbage collector

Vzhledem k tomu, že paměť není při „odstranění" objektů na haldě, které aplikace již nepotřebuje, neomezená, CLR používá techniku známou jako *Garbage Collection (GC*) [2].

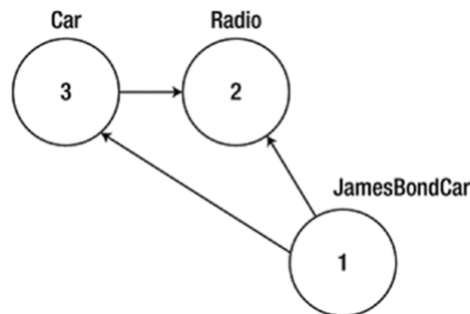
Sběr odpadu je zřejmě jednou z nejdůležitějších funkcí runtime, jehož účelem je obnovit paměť spotřebovanou objekty, na které již nejsou žádné odkazy. *Garbage collector* je zodpovědný jen a pouze za obnovu paměti a také určuje, co má být odstraněno na základě toho, zda zůstanou nějaké odkazy. Implicitně to znamená, že *Garbage collector* pracuje s referenčními objekty a obnovuje paměť pouze na haldě. Navíc to znamená, že zachování odkazu na objekt zpozdí sběr odpadků a také opakované použití paměti obsazené objektem [1].

Pro pochopení toho, jak *Garbage collector* určuje objekt, který již není potřeba, musíme znát *application roots* (*aplikační kořeny*). Jednoduše řečeno, kořenem je umístění úložiště, jež obsahuje odkaz na objekt umístěny v *Managed heap*, který může být součástí některé z následujících kategorií:

- referencí na globální objekty (i když nejsou povoleny v jazyce C#, kód CIL povoluje alokaci globálních objektů),
- referencí na statické objekty nebo statické parametry,
- odkazy na lokální objekty v rámci *codebase* aplikace,
- referencí na parametry objektu, které byly předané do metody,
- odkazy na objekty čekající na *finalizaci*,
- libovolný registr CPU, který odkazuje na objekt.

Během procesu sběru odpadu bude runtime zkoumat objekty na *Managed heap* za účelem určení, zda jsou aplikací stále dosažitelné (tj. *rooted*). K tomu CLR sestaví graf objektů (*object graph*), který reprezentuje každý dostupný objekt na haldě. Grafy objektů (Obr. 3) jsou

používány pro dokumentování všech dosažitelných objektů a nikdy se nevytváří pro stejný objekt dvakrát [3].



Obr. 3. Jednoduchý graf objektů [3, s. 783].

Před spuštěním *Garbage Collection* CLR nejprve pozastaví všechna vlákna v procesu. Tím zabraňuje vláknům přístup k objektům a změně jejich stavu, než je prozkoumá. Následně CLR provede operaci GC nazývanou *marking phase* (*fáze označení*), která spočívá v tom, že *Garbage collector* projde veškeré objekty na haldě a nastaví bit v *sync index* bloku objektu na 0. To znamená, že všechny tyto objekty mohou být odstraněny. Následně CLR kontroluje všechny aktivní kořeny a objekty, na které odkazují. Pokud kořen obsahuje *null*, CLR ho ignoruje a pokračuje k dalšímu kořenu [2].

Pokud má kořen odkaz na objekt, hodnota bitu v *sync index* bloku objektu bude změněna na 1 – to je *příznak označeného objektu*. Potom probíhá rekurzivní kontrola všech kořenů objektu a označení objektů, na které kořeny odkazují. Potká-li CLR již označený objekt, přerušuje rekurzi, aby v případě vzájemné reference mezi objekty nedošlo k nekonečné smyčce [2].

Po kontrole všech kořenů halda obsahuje sadu označených a neoznačených objektů. Označené objekty přežijí sběr odpadu, protože na ně odkazuje alespoň jeden objekt, tzn. jsou dosažitelné z aplikačního kódu. Neoznačené objekty jsou nedosažitelné, protože aplikace nemá kořen, přes který by k nim mohla mít přístup [2].

Když CLR ví, které objekty musí zůstat a které mohou být odstraněny, začíná další fáze sběru odpadu, nazývaná *compacting phase* (*fáze shlukování nebo komprese)*. V této fázi CLR přesune všechny objekty, které nejsou určeny k odstranění, tak aby byly umístěny v paměti bezprostředně za sebou [2].

Takový přesun objektů v paměti má následující výhody:

- Objekty se budou nacházet blízko od sebe, to vede ke snížení velikosti *working set size* (pracovní sady aplikace) a zlepšuje se tak výkon přístupu k těmto objektům v budoucnu.
- Volný prostor se také stává spojitým, což umožňuje uvolnit tuto oblast adresního prostoru.
- Komprese zabraňuje problémům fragmentace adresního prostoru, jak to bývá při použití *Managed heap* neboli *managed* kódu [2].

CLR při komprimaci paměti s objekty hýbe a to znamená, že objekty, které přežily sběr odpadu, odkazují na původní umístění v paměti, nikoliv v nové. A právě proto jako součást fáze shlukování CLR odečte od každého kořene počet bajtů, o které byl dotyčný objekt v paměti posunut. Tím je zajištěno, že každý kořen odkazuje na stejný objekt jako před fází shlukování [2].

Po shlukování paměti na *Managed heap* je první adresa po posledním objektu uložena do *NextObjPtr* (*Next Object Pointer*) a následně jsou spuštěné na začátku pozastavená vlákna v procesu. Pak aplikace přistupuje k objektům jako by GC vůbec neexistoval [2].

1.3.3.1 Generace objektů

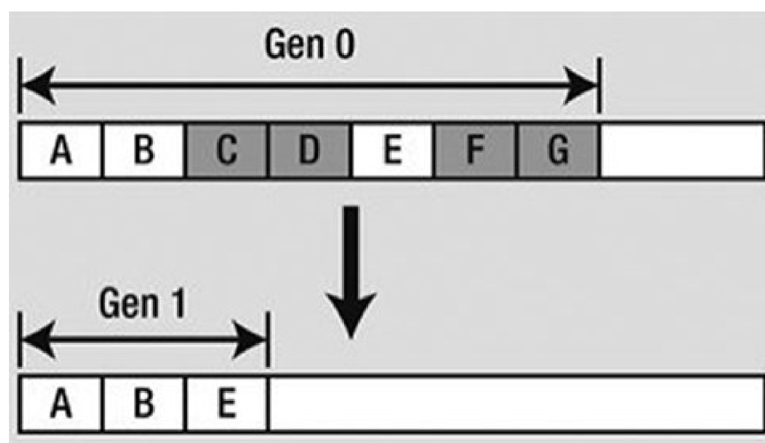
Garbage collector v CLR podporuje generaci objektů, je to tzv. *generational garbage collector*, známý také jako *ephemeral garbage collector* (*efemérní GC*), který vykonává svoji činnost podle následujících předpokladů:

- čím mladší je objekt, tím je kratší jeho životnost,
- čím starší je objekt, tím delší je jeho životnost,
- sběr odpadu v jedné části haldy je rychlejší než v celé haldě [2].

Ihned po inicializaci v *Managed heap* nejsou žádné objekty a objekty vytvořené v haldě představují generaci č. 0. To znamená, že nulová generace obsahuje jenom nově vytvořené objekty, na které zatím *Garbage collector* nesahal [2].

Po inicializaci CLR specifikuje prahovou hodnotu alokované paměti pro nulovou generaci. To znamená, že jakmile bude v důsledku alokování paměti pro objekt nulové generace tato hodnota překročena, musí začít sběr odpadků [2].

Ze všech objektů, které přežily sběr odpadků, se stávají objekty první generace, a také je zřejmé, že tyto objekty jsou již jednou zkontrolované. Tím pádem po GC nezůstanou žádné objekty nulové generace (Obr. 4). Dále CLR specifikuje prahovou hodnotu i pro generaci č.1 [2].



Obr. 4. Sběr odpadků v nulové generaci [3, s. 486].

Když velikost nulové generace znovu dosáhne prahové hodnoty, opět se spustí sběr odpadků. Tentokrát *Garbage collector* rozhoduje, které generace by měly být zpracovány.

Na začátku sběru odpadků GC zjišťuje objem paměti obsazené první generací objektů. Zatímco tato generace zabírá mnohem méně paměti, než je stanoveno, *Garbage collector* kontroluje jenom objekty nulové generace [2].

Podle prvního předpokladu „čím mladší je objekt, tím je kratší jeho životnost“ je velmi pravděpodobné, že v nulové generaci bude spousta odpadků a jejich promazání uvolní hodně paměti, zároveň je sběr odpadu značně urychlen, protože GC nezasahuje do první generace [2].

Výkon GC roste i díky selektivní kontrole objektů. Pokud má kořen nebo objekt odkaz na objekt ze starší generace, *Garbage collector* ignoruje všechny interní reference staršího objektu a to zkracuje dobu potřebnou k vytvoření grafu dostupných objektů. Samozřejmě je možné, že starý objekt odkazuje na nový. Aby nedošlo k vynechání aktualizovaných polí těchto starých objektů, používá GC interní mechanismus kompilátoru JIT, který nastavuje příznak při změně referenčního pole objektu [2].

Tento příznak umožňuje GC zjistit, které staré objekty (pokud nějaké existují) od posledního sběru odpadu byly změněny. Pak jenom zbývá zkontrolovat pouze staré objekty s upravenými poli a zjistit, zda odkazují na nové objekty z generace 0 [2].

Garbage collector s podporou generace předpokládá, že objekty, které žijí v paměti dost dlouho, budou žít i dále. Je tedy pravděpodobné, že objekty generace 1 budou i nadále dostupné v aplikaci. Také z toho plyne, že kontrolou objektů generace 1 by GC našel málo odpadků a nemohl by uvolnit mnoho paměti, to by nebylo vůbec racionální. Proto pokud se v generaci 1 objeví odpad, nějakou dobu tam zůstane [2].

V případě, že by se generace 1 rozrostla do takové míry, že by všechny její objekty společně překročily CLR stanovenou prahovou hodnotu, aplikace v tom okamžiku pokračuje v práci (protože sběr odpadků již byl dokončen) a standardně umísťuje objekty paměti, které zaplňují generaci 0, až do její prahové hodnoty [2].

Tím, že generace 0 dosáhne své prahové hodnoty, se inicializuje sběr odpadků. *Garbage collector* zjistí, že prostor obsazený objekty první generace překročil prahovou hodnotu. Po několika operacích sběru odpadů v generaci 0 je velmi pravděpodobné, že několik objektů v generaci 1 je již nedostupných. Proto nyní *Garbage collector* zkontroluje všechny objekty generací 1 a 0 [2].

Výsledně dojde k tomu, že všechny objekty generace 0, které přežily, jsou povýšeny na první generaci a objekty z první generace jsou umístěné v generaci 2. Jako vždy bezprostředně po sběru odpadků je generace 0 prázdná a v generaci 2 jsou objekty, které byly zkontrolovány GC alespoň dvakrát [2].

Operace sběru odpadu můžou probíhat mnohokrát, ale objekty generace 1 jsou kontrolovány pouze v případě, že jejich celková velikost dosáhne prahové hodnoty – před tím se obvykle provádí několik operací sběru odpadu v generaci 0. Spravovaná halda podporuje pouze tři generace: 0, 1 a 2. Během inicializace CLR nastaví práh pro všechny tři generace [2].

Garbage collector v CLR je *self-tuning* (přizpůsobivý nebo samoladicí), to znamená, že v průběhu své práce analyzuje, kolik paměti bylo uvolněno a kolik objektů zbývá, pak v závislosti na přijatých datech může zvýšit nebo snížit prahové hodnoty pro všechny tři generace [2].

Pokud například aplikace vytvoří mnoho objektů a použije je po velmi krátkou dobu, znamená to, že úklid v generaci 0 vrátí hodně paměti. Současně, když bude zjištěno, že úklid odpadků v generaci 0 přežilo velmi málo objektů, GC může snížit prahovou hodnotu pro tuto generaci. V takovém případě bude sběr odpadků prováděn častěji a zároveň bude vyžadovat méně práce pro *Garbage collector*, protože *process's working set (pracovní sada procesů)* bude malý. V případě, že všechny objekty v generaci 0 jsou odpadky, zřejmě odpadá

i nutnost provádět komprimaci dat a bude postačující vrátit ukazatel *NextObjPtr* na začátek generace 0. Sběr odpadků pak lze považovat za dokončený [2].

Pokud však po sběru odpadků v generaci 0 zůstává mnoho objektů, znamená to, že se uvolnilo málo paměti. V tomto případě může *Garbage collector* zvýšit prahovou hodnotu pro generaci 0, to znamená, že sběr odpadků probíhá méně často a zároveň je uvolněno značné množství paměti [2].

1.3.4 Velké objekty

Doposud byly probrány pouze malé objekty, ale CLR rozděluje objekty na malé a velké. Tento fakt lze využít pro zvýšení rychlosti. Všechny objekty s velikostí 85 000 bajtů nebo více jsou považovány za velké. V budoucnu se ale tato hodnota může změnit, proto je lepší ji nepovažovat za konstantní [2].

CLR s velkými objekty pracuje podle mírně odlišných pravidel:

1. Paměť je pro CLR přidělena v samostatné části adresního prostoru procesu.
2. Komprese není aplikována na velké objekty, protože jejich přesunutí v paměti by zabralo příliš mnoho procesorového času. Možná fragmentace adresního prostoru mezi velkými objekty může způsobit výjimku *OutOfMemoryException*. Dále se může stát, že v příštích verzích CLR se budou velké objekty také účastnit procesu komprese.
3. Velké objekty jsou vždy považovány za součást generace 2, takže by měly být vytvořeny pouze pro zdroje, které by měly existovat dlouho. Umístění velkých objektů s krátkou životností v paměti by si v důsledku vyžádalo častý sběr odpadu druhé generace a tím by se snižovala rychlost aplikace. Obvykle jsou ve velkých objektech uložené velké textové řetězce (například XML nebo JSON) nebo bajtová pole, které jsou použité v I/O operacích, například při čtení dat ze souboru nebo sítě do vyrovnávací paměti pro následné zpracování [2].

Všechny tyto mechanismy jsou pro developera zcela transparentní. Jednoduše je možné na jejich existenci zapomenout, dokud v programu nenastane nějaká abnormální situace (například fragmentace adresního prostoru) [2].

1.3.5 Marshalling

Nezávisle na používaném *unmanaged* API je interoperabilita spojena s využitím speciálního vlákna, které v případě volání nativní funkce protíná hranici mezi řízeným a nativním kódem a pak zpět v případě návratu volání [4].

Marshalling je proces konverze dat v paměti k potřebnému typu pro další použití. Tento mechanismus se používá kvůli rozdílům ve způsobu reprezentace určitých typů v *managed* a *unmanaged* prostředí. Například řízený typ *System.Boolean* (*bool*) může mít různé reprezentace v unmanaged kódu: *bool* typ Win32 zabírá čtyři bajty a hodnotě *true* odpovídá libovolná nenulová hodnota, zatímco v C++ je hodnota typu *bool* reprezentovaná jedním bajtem a je to celé číslo 1 [4].

Vzhledem k tomu, že *marshalling* má značný vliv na rychlost aplikace, je potřeba v případě používání *unmanaged* API dbát na faktory, které způsobují zpomalení [4]. Například je dobré vědět o tzv. *blittable types (binárně kompatibilní typy),* typech, které mají stejnou reprezentaci v *managed* a *unmanaged* paměti a nevyžadují speciální zpracování za pomoci *marshallingu*. To znamená, že *blittable* typy nevyžadují konverzi, když jsou předávány mezi řízeným a neřízeným kódem, a také to, že jiné typy mohou být nejednoznačné nebo nejsou v řízené paměti vůbec reprezentovány [5].

Jednorozměrná pole s prvky binárně kompatibilních typů, kde všechny prvky patří ke stejnému typu, jsou také binárně kompatibilní, stejně jako struktury nebo třídy, které se skládají pouze z binárně kompatibilních typů [4].

Pro organizaci správného *marshallingu* binárně nekompatibilních argumentů musí CLR vědět, jakou konkrétní reprezentaci má použít [4]. Za tímto účelem se používají explicitní instrukce pro *marshalling* [6]. Obecně jsou veškeré potřebné metody, typy apod. v prostředí .NET reprezentovány v *namespace System.Runtime.InteropServices* [4].

Všechny *Blittable* typy používané v .NET (*namespace System*) (Tab. 1).

Tab. 1. Blittable typy C# a blittable nativní typy [7].

C# Type	**C Type**	**<stdint.h> Type**	**<glib.h> Type**
sbyte	char	int8_t	gint8
byte	unsigned char	uint8_t	guint8
short	short	int16_t	gint16
ushort	unsigned short	uint16_t	guint16
int	int long 32-bit platforms only	int32_t	gint32
uint	unsigned int unsigned long 32-bit platforms only	uint32_t	guint32
long	long 64-bit platforms only __int64 (MSVC) long long (GCC)	int64_t	gint64
long	unsigned long (64-bit platforms only) unsigned __int64 (MSVC) unsigned long long (GCC)	uint64_t	guint64
char	unsigned short	uint16_t	guint16
float	float		gfloat
double	double		gdouble

1.3.6 Uvolnění zdrojů pomocí mechanizmu finalizace

Je důležité zmínit také typy, které používají systémové prostředky nebo unmanaged kód.

Pokud typ používající systémový prostředek bude zničen během shromažďování odpadu, paměť, která byla obsazená objektem, se vrátí zpátky na *Managed heap*, avšak systémový nebo *unmanaged* (neřízený) prostředek, o kterém *Garbage collector* neví, bude ztracen. To je samozřejmě nežádoucí, a proto CLR podporuje mechanismus finalizace, který umožňuje dotyčnému objektu provést korektní uvolnění obsazených prostředků předtím, než *Garbage collector* uvolní obsazenou paměť. Každý typ, který používá systémový prostředek (soubor, síťové připojení, soket, *mutex* atd.), musí podporovat finalizaci [2].

Jakmile CLR zjistí, že objekt už není dostupný, je mu dána možnost provést finalizaci s uvolněním všech použitých systémových prostředků, po němž je objekt vrácen na *Managed*

heap. Obecná základní třída *System.Object* definuje chráněnou virtuální metodu jménem *Finalize*. Když *Garbage collector* detekuje, že objekt má být zničen, volá nad objektem metodu *Finalize,* (pokud je předefinována) [2].

Chceme-li tedy naimplementovat finalizační metodu v jazyce C#, musíme před název třídy přidat znak tildy (~) (Obr. 5).

```
internal sealed class SomeType {
   // This is the Finalize method
   ~SomeType() {
      // The code here is inside the Finalize method
   }
}
```

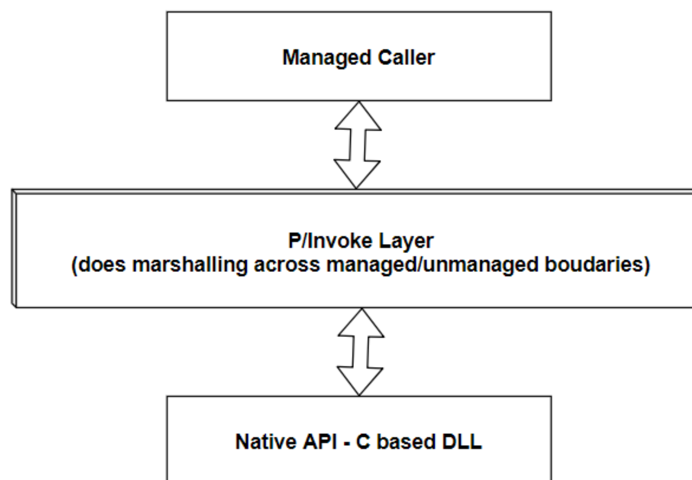
Obr. 5. Finalizační metoda [2, s. 525].

2 ZPŮSOBY VYUŽITI UNMANAGED KÓDU V JAZYCE C#

Tato kapitola se zabývá způsoby využití unmanaged DLL knihoven pomocí jazyků C# nebo technologii C++/CLI, ukazuje, jak lze pomocí pointerů operovat s pamětí v rámci CLR.

2.1 Platform Invoke

Mechanismus *Platform Invoke*, známý také jako *P/Invoke*, umožňuje v řízeném prostředí vyvolávat funkce ve stylu jazyka C/C++, které jsou vyexportované v DLL knihovně [1]. Všechny potřebné položky a API *P/Invoke* jsou obsaženy ve dvou jmenných prostorech: *System* a *System.Runtime.InteropServices* [8]. Samotný mechanismus *P/Invoke* je zobrazen na obrázku (Obr. 6).



Obr. 6. Mechanismus P/Invoke [9, s. 158].

Pro používání mechanismu *P/Invoke* v *managed* kódu je nutné mít deklaraci statické externí (*static extern*) metody se signaturou ekvivalentní funkci v jazyce C/C++. K samotné metodě musí být přidán atribut *DllImport* definující alespoň DLL knihovnu, která požadovanou funkci exportuje [1].

V kontextu *P/Invoke* klíčové slovo *extern* indikuje kompilátoru, že implementace metody se nachází v externí knihovně a v podstatě se používá jenom v kombinaci s atributem *DllImport* nebo při práci s COM [4].

Atribut *DllImport* umožňuje kódu .NET volat funkce libovolné *unmanaged* knihovny napsané v jazyce C nebo C ++, anebo nízkoúrovňové API operačního systému [3] (Obr. 7).

```
using System;
using System.Runtime.InteropServices;

public class Program
{
    // Import user32.dll (containing the function we need) and define
    // the method corresponding to the native function.
    [DllImport("user32.dll", CharSet = CharSet.Unicode, SetLastError = true)]
    public static extern int MessageBox(IntPtr hWnd, string lpText, string lpCaption, uint uType);

    public static void Main(string[] args)
    {
        // Invoke the function as a regular managed method.
        MessageBox(IntPtr.Zero, "Command-line message box", "Attention!", 0);
    }
}
```

Obr. 7. Používání atributu DllImport [8].

2.2 C++/CLI

C++/CLI je technologie, která slouží jako propojovací prvek mezi programovacím jazykem *Standard C++* a *Common Language Infrastructure (CLI).* Toto podání se vyvinulo z jiného projektu společnosti Microsoft, *Managed Extensions for C++,* jehož první široce distribuovaná implementace byla vydána společností v červenci 2000 jako součást iniciativy .NET Framework. První široce distribuovaná beta implementace C++/CLI byla vydána v červenci 2004 [10].

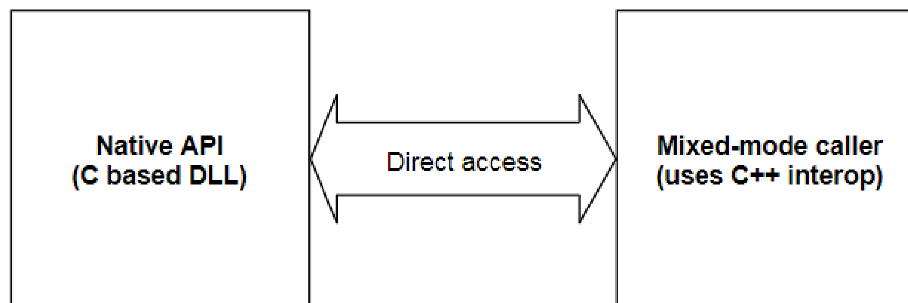
Návrh standardu C++/CLI odpovídá následujícím cílům:

- Poskytovat elegantní a jednotnou syntaxi a sémantiku, které dávají programátorům C++ přirozený pocit.
- Poskytovat prvotřídní podporu funkcionalit CLI (např. *properties, events, garbage collection, and generics*) pro všechny typy včetně stávajících tříd Standardu C++.
- Poskytovat prvotřídní podporu pro standardní funkce C++ (např. *deterministic destruction, templates*) pro všechny typy včetně tříd CLI.
- Zachovat význam stávajících aplikací Standardu C++ specifikováním čistých rozšíření všude, kde je to možné [10].

Obecně platí, že C++ *Interop* má lepší výkon než *P/Invoke*, protože má méně požadavků na marshalling typů. Ačkoliv lze *P/Invoke* také používat efektivně pomocí vlastní implementace marshallingu. C++ *Interop* ale poskytuje větší kontrolu nad konverzí *managed* a *unmanaged* typů a dovoluje zrychlit volání do nativního kódu pomocí *unmanaged* bloků ve zdrojovém kódu [9].

2.2.1.1 C++ Implicit Interop

Potřebujeme-li získat přístup k nativní knihovně pomocí C++ *Interop*, stačí pomocí direktivy *#include* přidat požadované *header* soubory a nastavit referenci na související **.lib* soubory. Obrázek č. 8 ukazuje, jak lze v *mixed-mode* (smíšeném režimu) přímo přistupovat k nativnímu rozhraní API prostřednictvím C++ *Interop,* a to zcela bez prostředníka [9].



Obr. 8. Přístup k nativní knihovně pomocí C++ Interop [9, s. 160].

2.3 Klíčové slovo „unsafe“

Jednou ze skvělých vlastností C# je skutečnost, že je silně typovaný a podporuje kontrolu typu za běhu runtime. Obzvláštní výhodnou je i to, že tuto vlastnost lze obejít manipulováním s pamětí a adresami v ní. Hodilo by to například při práci s paměťovými zařízeními nebo při implementaci časově náročných algoritmů. Klíčem *unsafe (nebezpečný)* lze označit část kódu jako nebezpečnou. Nebezpečný kód je explicitní blok kódu a zároveň opce kompilace. Modifikátor nemá žádný vliv na vygenerovaný CIL kód, ale je to direktiva pro kompilátor, která umožňuje manipulaci s ukazateli a adresami v rámci *unsafe* bloků. Kromě toho *unsafe* neznamená unmanaged [1] (Obr. 9).

```
class Program
{
  static void Main(string[] args)
  {
    unsafe
    {
      // Work with pointer types here!
    }

    // Can't work with pointers here!
  }
}
```

Obr. 9. Unsafe blok [3, s. 438].

Kromě deklarování rozsahu nebezpečného kódu je v rámci metody možné také vytvářet „nebezpečné“ struktury, třídy, členy typu a parametry. Po vytvoření nebezpečného kontextu pak lze libovolně pomocí operátoru „*“ vytvářet ukazatele na datové typy a pomocí operátoru „&“ získávat jejich adresy. Na rozdíl od C nebo C++ je v C# operátor „*“ aplikován pouze na základní typ, nikoli jako předpona ke každému názvu proměnné ukazatele [3] (Obr. 10).

```
// No! This is incorrect under C#!
int *pi, *pj;

// Yes! This is the way of C#.
int* pi, pj;
```

Obr. 10. Deklarace pointerů v C# [3, s. 440].

3 DATA

Moderní digitální počítač byl vynalezen a určen jako zařízení, které by mělo usnadnit a urychlit komplikované a časově náročné výpočty. Většinou je ale za jeho primární charakteristiku považována schopnost ukládat a přistupovat k velkému množství informací a jeho výpočetní schopnost, tzn. provádět aritmetické operace, se naopak v často stává téměř irelevantní. V obou případech představuje velké množství informací, které mají být zpracovány, určitou abstrakci části reálného světa. Informace, které má počítač k dispozici, se skládají z vybraného souboru dat o reálném světě, konkrétně ze souboru, který je považován za relevantní pro konkrétní řešený problém [11, s. 1].

Data tedy představují abstrakci skutečnosti a to znamená, že některé vlastnosti a charakteristiky reálných objektů jsou ignorovány, protože jsou periferní a irelevantní pro zkoumaný problém. Abstrakce je tudíž také zjednodušením faktů o reálném objektu [11, s. 1].

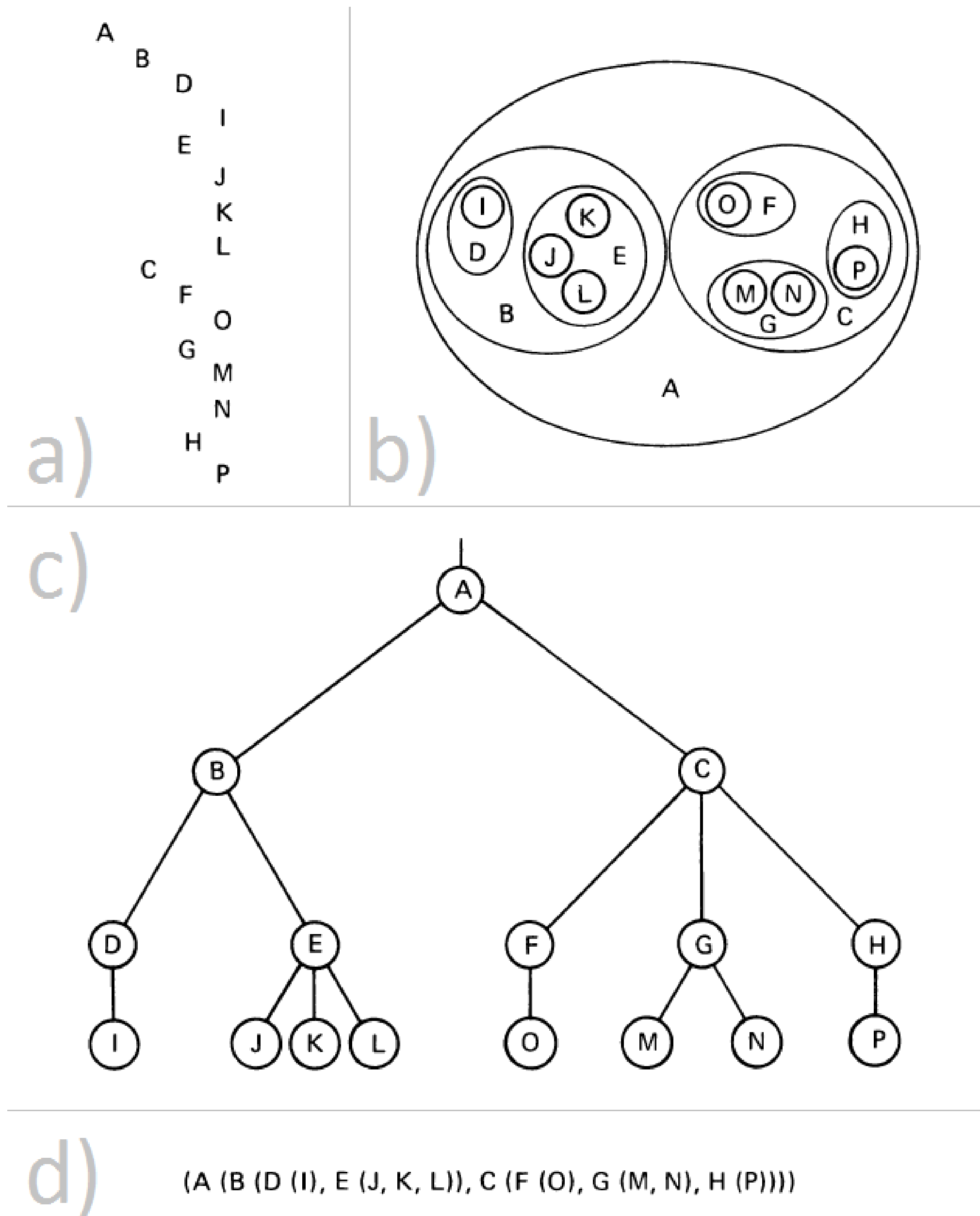
3.1 Stromové struktury

Strom představuje hierarchickou strukturu postavenou nad kolekcí položek. Známými příklady stromů jsou rodokmeny a organizační diagramy. Dále lze stromy použit k analýze elektrických obvodů nebo k reprezentaci struktury matematických vzorců. Stromové struktury také vznikají přirozeně ve mnoha různých oblastech informatiky (Obr. 11). Stromy se například používají k uspořádání informací v databázových systémech anebo k reprezentaci syntaktické struktury zdrojového kódu v kompilátorech [12].

Strom je kolekce prvků zvaných *uzly*, jeden z uzlů představuje *root (kořen*) spolu s relací „rodičovství“ neboli „*parenthood*“, která umísťuje hierarchickou strukturu na uzlech. Uzel může být libovolného typu [12].

Formálně lze strom definovat rekurzivně následujícím způsobem:

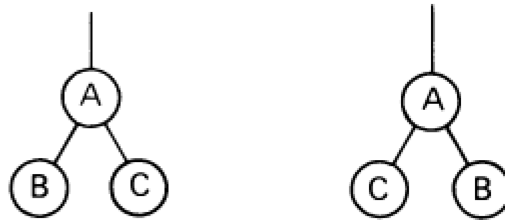
1. Samotný uzel je strom a zároveň také kořen stromu.
2. Předpokládejme, že n je uzel a $T_1, T_2, \ldots, T_k$ jsou stromy s kořeny $n_1, n_2, \ldots, n_k$. Můžeme vytvořit nový strom tak, že n bude rodičem uzlů $n_1, n_2, \ldots, n_k$. V tomto stromě n je kořen a $T_1, T_2, \ldots, T_k$ jsou *podstromy (subtrees)* kořene. Uzly $n_1, n_2, \ldots, n_k$ se nazývají *děti (children)* uzlu n [12].



Obr. 11. Zobrazení stromových struktur: a) odsazení (indentation); b) vnořené množiny; c) graf; d) vnořené závorky [11, s. 190].

Uspořádaný *binární strom* definujeme jako konečnou množinu prvků (*uzlů*), která je buď prázdná, nebo se skládá z kořene (uzlu) se *dvěma* nepropojenými binárními stromy, které reprezentují levý a pravý podstrom kořene [11].

Strom je *dokonale vyvážený,* pokud se pro každý uzel stromu počet uzlů v levém a pravém podstromu liší nejvýše o 1 uzel [11] (Obr. 12).



Obr. 12. Dva odlišné binární stromy [11, s. 191].

3.2 AVL b-tree

Vzhledem k tomu, že obnovení dokonalé rovnováhy po náhodném vložení je poměrně složitá operace, možným vylepšením je formulace méně přísných definic „rovnováhy“, které by měly vést k jednodušším postupům při reorganizaci stromu za cenu pouze mírného zhoršení průměrné výkonnosti vyhledávání [11].

Jedna taková definice rovnováhy byla zformulovaná Adeľsonem-Veľskim a Landisem, podle nichž je kritérium rovnováhy chápáno takto: Strom je vyvážen pouze tehdy, když pro každý uzel stromu platí, že rozdíl mezi výškou jeho dvou podstromů není větší než 1 [11].

Stromy splňující tuto podmínku se často nazývají AVL stromy (podle jejich vynálezců). Definice těchto stromů je nejen jednoduchá, ale také nabízí jednodušší způsob vyvažování a průměrnou délku cesty při vyhledávání, která je prakticky shodná s délkou *dokonale vyváženého stromu.* Takové operace jako vyhledávání, vkládání nebo odstranění uzlu na vyvážených stromech probíhají v logaritmicky omezeném čase O (log n), a to i v nejhorším případě [11].

II. PRAKTICKÁ ČÁST

4 IMPLEMENTACE

Tato kapitola popisuje postup a techniky použité v počítačovém kódu, který je předmětem praktické části práce. Použité algoritmy cashování textových řetězců jsou představeny jako pár klíč-text, aby byl *behchmark* co nejvíce přiblížený reálnému světu. Využití algoritmu *unblittable* typu *String* pak vede při komunikaci mezi řízeným a neřízeným kódem (a opačně) k používání *marshallingu.*

Pro všechny implementace managed kódu je použit .NET Framework verze 4.7.2. Vývoj včetně testování probíhal na stroji s procesorem *Intel Core i5* rodiny *Ivy Brige* s 4 GB DDR3 paměti a operačním systémem *Windows 7 Professional*. Kvůli menšímu objemu paměti jsou všechny projekty zkompilované do 32bitové architektury, to znamená, že také C/C++ kompilátory provádí optimalizaci s ohledem na tuto skutečnost.

4.1 Binární strom

Binární strom se používá jako vzorový algoritmus ve všech použitých technologiích. Samotná implementace AVL binárního stromu je inspirovaná algoritmy, které popsal švýcarský informatik Niklaus Wirth v roce 1976.

4.2 Interop

V této části práce jsou popsány způsoby využití managed kódu v řízeném prostředí CLR a zároveň jsou prezentována řešení pro různé realizace *Interop* s krátkým popisem výhod a nevýhod konkrétní technologie.

4.2.1 P/Invoke

V testovacím projektu technologie *P/Invoke* se počítá s nativními knihovnami v jazycích C a C++.

Nejsilnější stránkou *P/Invoke* je to, že danou technologii lze využít v jiných operačních systémech než Windows, tzn. že je *cross platform*. Do června 2016 šlo používat tuto technologii v operačních systémech MacOS a Linux pouze s platformou MONO[1]. Pak Microsoft vydal crossplatformní edici .NET zvanou *.NET CORE*, kterou lze používat i v Linux nebo MacOS.

[1] Více informací viz https://www.mono-project.com/docs/advanced/pinvoke/

Nevýhoda se skrývá v samotné technologii *P/Invoke*, která dovoluje volat jenom určité vyexportované funkce nebo metody, tzn. že nejde používat třídy stejným způsobem, jak tomu je v C#, například nelze volat metody nebo funkce, které vracejí objekt. Také vyexportované nativní funkce nejde volat bez použití atributu *DllImport*, což do jisté míry znepřehlední kód (Obr. 13).

4.2.1.1 P/Invoke Interop Asistent

Vzhledem k tomu, že třída marshallingu v .NET obsahuje hodně atributů a pravidel, její pochopení může zabrat hodně času a za určitých okolností způsobit i frustraci. Například v případě, kdy signatura volané nativní funkce obsahuje jiný typ než *blittable* a vývojář z nějakého důvodu nechce implementovat vlastní *marshalling*, určitou pomoc pro správné generování potřebných atributů poskytuje nástroj *P/Invoke Interop Asistent*[2].

```
/// <summary>Add string and key to the tree. Use native func _Add(int key, wchar*).</summary>
/// <param name="data">wchar_t* text string.</param>
/// <param name="key">unsigned int 'Key'. Must be unique.</param>
[DllImport(@"AvlBinTree_Cpp_VS_lib.dll")]
private static extern void _Add
    (int key,
        [System.Runtime.InteropServices.InAttribute()]
        [System.Runtime.InteropServices.MarshalAsAttribute(
            System.Runtime.InteropServices.UnmanagedType.LPWStr)] string data);
```

Obr. 13. Vygenerované atributy pomocí P/Invoke Interop Asistent.

Výše zmíněný nástroj mimo jiné pomáhá i při vývoji s použitím nativních knihoven operačního systému *Windows*.

4.2.1.2 C DLL

Pro vytvoření této knihovny byl použit *cross platform* IDE Code Blocks ve verzi 17.12. Pro kompilaci pak GNU GCC kompilátor verze 5.1.0 s flagy: /flto /mfpmath=sse /Ofast /arch=native /O2 /m32.

Managed kód pro ukládání textu používá objekt *String*, který je v .NET reprezentován posloupností kódových jednotek UTF-16, a zároveň je takovou jednotkou v operačním sytému

[2] Zdrojové kódy jsou uvedeny na GitHub: https://github.com/jaredpar/pinvoke-interop-assistant

Windows nativní typ *wchar_t*. Proto bylo rozhodnuto použít pro reprezentaci textu v nativním kódu pole *wchar_t (respektive wchar_t*)*. Právě toto rozhodnutí přineslo méně práce s přetypováním při překročení hranice mezi řízeným a neřízeným kódem.

Tato implementace vůbec nepoužívá .NET Interop třídu *Marshal*. Na obrázku 14 je zobrazena řízena metoda *Add*, která používá klíčové slovo *unsafe*, a to právě proto, aby v dalších krocích předala do nativní funkce *Add* zafixovaný pointer s délkou vkládaného řetězce. Samozřejmě i klíč, který ale vzhledem k tomu, že je *blittable*, nepředstavuje žádnou komplikaci, a proto není tak zajímavý.

```
public void Add(int key, string data)
{
    unsafe
    {
        fixed (char* pData = data)
        {
            Add(key, (IntPtr)pData, data.Length);
        }
    }
}
```

Obr. 14. Klicová slova unsafe a fixed.

Na obrázku 15 je vidět nativní procedura, která používá zaslaný pointer na *String*, jenž se nachází stále v řízené paměti, ke kopírování tohoto textu do neřízené paměti pomocí nativní funkce *wmemcpy*. Právě proto je nezbytně nutné v řízeném kódu používat *statement* (prohlášení) *fixed*, které zabraňuje GC provádět jakoukoli manipulaci s tímto textovým řetězcem.

```
void DLL_API Add(unsigned int key, const wchar_t *data, unsigned int len)
{
    if (key < 0) {
        return;
    }
    wchar_t* newData = (wchar_t*)malloc(sizeof(wchar_t) * len);
    _root = _Insert(newNode(key, wmemcpy(newData, data, len)), _root);
    newData = NULL;
}
```

Obr. 15. Nativní procedura Add.

Při vracení textového řetězce zpět z neřízené paměti do řízené není potřeba právě kvůli používání typu *wchar_t* používat *marshalling* a rovnou lze využít pointer na pole uložené v *unmanaged* paměti. Jak lze vidět na obrázku 16, tento pointer se pak používá přímo v konstruktoru řízeného objektu *String*.

```
public string GetText(int key)
{
    string result;
    unsafe
    {
        result = new String((char*)FindData(key));
    }
    return result;
}
```

Obr. 16. Managed metoda GetText.

Destruktor je reprezentován vyexportovanou bezparametrickou procedurou *DestroyTree()*, která následně zavolá interní rekurzivní proceduru (Obr. 17).

```
void DLL_API DestroyTree()
{
    if (destructorIsRunning)
        return;
    destructorIsRunning = 1;
    _DestroyTree(_root);
    _root = 0;
}
void _DestroyTree(struct node* n)
{
    if (n == 0) return;
    _DestroyTree(n->left);
    _DestroyTree(n->right);
    free(n->data);
    free(n);
}
```

Obr. 17. Nativni procedura-destructor.

4.2.1.3 C++ Dll

Pro vytvoření této knihovny C++ bylo použito IDE *Visual Studio 2017*. Pro kompilaci byl použit *Intel C++ Compiler* verze *19.0*, který se poskytuje se sadou nástrojů *Intel Parallel Studio XE 2019*. Také byly použity následující optimalizační flagy: /Oi /O3 /Ot /QxHost /Quse-intel-optimized-headers.

Jak je vidět na obrázku 18, princip marshallingu je stejný jako v knihovně napsané v jazyku C, a to s jediným rozdílem, že se zde používá klíčové slovo *extern „C“*, potřebné pro volání vyexportovaných metod v jazyce C++ a tím pádem i C++/CLI.

```
#ifdef AVLBINTREECPPVS_EXPORTS
#define BINTREE_API __declspec(dllexport)
#else
#define BINTREE_API __declspec(dllimport)
#endif

extern "C" BINTREE_API void _Init();
extern "C" BINTREE_API void  _Add(unsigned int key,
                                   const wchar_t *data, unsigned int len);
extern "C" BINTREE_API const wchar_t* _FindData(unsigned int key);
extern "C" BINTREE_API void _DestroyTree();
```

Obr. 18. Nativní C++ API.

Pro komunikaci s touto knihovnou přímo v testovacím projektu byla vytvořena třída *BTree_Cpp_msvc_dll*, která pomocí atributu *DllImport* volá vyexportované funkce nativní dll. Tato třída má také kvůli bezpečnému promazání dat v nativní paměti realizovaný interface *System.IDisposable*.

4.2.2 C++/CLI

Technologie C++/CLI v testovacím projektu byla použita dvěma způsoby, které jsou popsané v jednotlivých podkapitolách.

Nejsilnější výhodou C++/CLI je to, že tato technologie dovoluje vývojáři používat veškeré možnosti jak *managed*, tak i *unmanaged* kódu zároveň, a proto je jazyk nejlepší volbou pro napsání nativních *wrappers (adaptérů)*. Je to také jediná technologie v .NET, která dovoluje zcela přirozeně a bez sebemenších problémů používat nativní třídy v řízeném prostředí CLR.

Nevýhodou naopak je to, že tuto technologii lze používat pouze v operačním systému Windows, což znamená, že C++/CLI zatím není *cross platform*. Pozitivním faktem je, že probíhá práce na implementaci C++/CLI do edice .NET CORE[3].

Pro vývoj bylo použito IDE *Visual Studio 2017* a *Visual C++* kompilátor.

4.2.2.1 C++/CLI

Jak již bylo zmíněno výše, C++/CLI perfektně spojuje *managed* a *unmanaged* kód a tato vlastnost je použita i v testovacím projektu. V našem konkrétním případě se *managed* C++/CLI a *unmanaged* C++ kód nachází ve stejné *assembly* (knihovně).

---

[3] Aktuality sledovat lze v tomto vlákně na GitHub: https://github.com/dotnet/coreclr/issues/18013

Na obrázku 19 je zobrazená *managed* třída *CppCliBTreeAPI*, která poskytuje rozhraní pro komunikaci CLR s nativní třídou *AvlBinTreeNative*. Oproti výše popsaným implementacím je vidět, že toto rozhraní neboli *wrapper* (obálka) pracuje přímo s *managed* objektem typu *String*.

```
#pragma managed
namespace CppCliNativeBTree {
    public ref class CppCliBTreeAPI
    {
    public:
        CppCliBTreeAPI();
        ~CppCliBTreeAPI();
        !CppCliBTreeAPI();

        void Add(int key, System::String ^ data);
        System::String^ FindData(int key);
    private:
        AvlBinTreeNative *treeNativePtr;
    };
}
```

Obr. 19. C++/CLI API.

Na obrázku 20 jsou zobrazené public metody *unmanaged* C++ třídy *AvlBinTreeNative*. Podle kódu na obrázku si lze všimnout, že do metody *Add* se už neposílá délka textového řetězce. Skutečnost, že třída má být zkompilovaná do nativního kódu, naznačuje direktiva *#pragma unmanaged.*

```
#pragma unmanaged
class AvlBinTreeNative
{
public:
    AvlBinTreeNative();
    ~AvlBinTreeNative();
    void Add(unsigned int key, const wchar_t *data);
    const wchar_t* FindData(unsigned int key);
```

Obr. 20. Nativní kód v C++/CLI assembly.

Jak je vidět na obrázku 21, v této implementaci se používá marshalling pomocí *managed* třídy *Marshal*, která za pomoci metody *StringToHGlobalUni* kopíruje textový řetězec do *unmanaged* paměti a vrátí pointer na tato zkopírovaná data pro další použití v *unmanaged* kódu. Zároveň je na obrázku zobrazena metoda *FindData*, která pomocí referenčního objektu *gcnew* vytváří v *managed* paměti nový *String* objekt.

```
void CppCliBTreeAPI::Add(int key, System::String ^ data)
{
    using namespace System::Runtime::InteropServices;
    treeNativePtr->Add(key,
        (const wchar_t*)(Marshal::StringToHGlobalUni(data)).ToPointer());
}

System::String^ CppCliBTreeAPI::FindData(int key)
{
    return gcnew System::String(treeNativePtr->FindData(key));
}
```

Obr. 21. C++/CLI Marshalling.

Vzhledem k tomu, že se používaná data nachází v neřízené paměti, je v managed třídě *CppCliBTreeAPI* naimplementován destruktor a finalizátor, které pomocí operátoru *delete* volají destruktor nativní třídy *CppCliBTreeAPI* (Obr. 22). Pak pomocí rekurzivní metody *_DestroyTree* budou promazané jak datové struktury vytvořené v nativním kódu, tak i nativní data vytvořená přímo z prostředí CLR.

```
AvlBinTreeNative::~AvlBinTreeNative()
{
    if (destructorIsRunning)
        return;
    destructorIsRunning = true;
    _DestroyTree(_root);
    _root = 0;
}

void AvlBinTreeNative::_DestroyTree(node* n)
{
    if (n == 0) return;
    _DestroyTree(n->left);
    _DestroyTree(n->right);
    delete n->data;
    delete n;
}
```

Obr. 22. C++/CLI destruktor.

4.2.2.2 C++/CLI mixed mode

Další výhodou technologií C++/CLI je možnost využití tzv. *implicit C++ Interop*. V rámci experimentu bylo rozhodnuto použít výše popsanou C++ dll právě s touto technologií, která se běžně používá pro komunikaci mezi C a C++ knihovnami.

Bonusovou výhodou při použití C++ *Interop* v C++/CLI je fakt, že při komunikaci mezi *managed* a *unmanaged* kódem nevzniká žádná mezivrstva, jak je tomu při použití technologie *P/Invoke*. Proto je tato komunikace pro vývojáře více transparentní. Další výhodou oproti předchozí realizaci je možnost použití optimalizace C++ kompilátoru, a to z toho důvodu, že nativní kód je vyexportován do samostatné DLL. Jako nevýhoda může být vnímána skutečnost, že tímto způsobem vznikají dvě knihovny místo jedné.

Na obrázku 23 je zobrazená implementace *managed* třídy, která pomocí direktivy *#include* obsahuje hlavičku nativního API. Tato hlavička se dotahuje přímo z nativního projektu pomocí cesty nastavené v sekci *Additional Include Directories*. Dále je pro kompilaci projektu používajícího C++ *Interop* nezbytně nutné nakonfigurovat i *Linker*. A to proto, že právě při kompilaci musí být načtena statická knihovna *lib*, která vzniká při kompilaci nativní DLL. Takže *Linker* potřebuje znát cestu ke knihovně *lib* a znát její název. Vše se nastavuje v konfiguraci projektu, a to v polích *Additional Library Directories* a *Additional Dependencies*.

```
#include <vcclr.h>
#include "BinTreeAPI.h"
using namespace System;

namespace AvlBinTreeCppCliImplicitInteropCpp {
    public ref class CppCliImlicitInteropCpp
    {
    public:
        CppCliImlicitInteropCpp();
        ~CppCliImlicitInteropCpp();
        !CppCliImlicitInteropCpp();
        void Add(int key, System::String^ data);
        System::String^ Find(int key);
    };
}
```

Obr. 23. C++/CLI mixed mode (implicit C++ Interop).

Tato implementace nepoužívá třídu *Marshal*, ale funkci *PtrToStringChars*, která se nachází v hlavičce *vcclr.h* a slouží pro vytvoření GC pointeru ze řízeného objektu *String* (Obr. 24).

```
void CppCliImlicitInteropCpp::Add(int key, System::String^ data)
{
    pin_ptr<const wchar_t> pWch = PtrToStringChars(data);
    _Add(key, pWch, data->Length);
}
```

Obr. 24. C++/CLI vytvoření pointeru na textový řetězec.

Pro textové řetězce do řízené paměti se používá konstruktor *managed* objektu *String*.

4.2.3 C# managed

C# implementace AVL binárního stromu je čistě *managed* generická třída s interface *IComparable*, která je inspirovaná použitým fundamentálním C++/C algoritmem s ohledem na omezení řízeného prostředí CLR. Tuto konkrétní implementaci lze tedy považovat za nativní. Jednou z jistých nevýhod je, že jako uzel stromu je použita třída místo struktury (Obr. 25).

```
public class Node
{
    public K Key;
    public int Height;
    public Data<T> Data;
    public Node Left;
    public Node Right;

    public Node(K key, Data<T> data)
    {
        this.Key = key;
        this.Height = 1;
        this.Data = data;
        this.Left = null;
        this.Right = null;
    }
}
```

Obr. 25. C# uzel stromu.

V testovacím projektu je jako představitel binárních stromů použita třída *SortedDictionary*, která je implementována jako *Red-black tree*[4].

[4] Implementaci lze prostudovat zde: https://referencesource.microsoft.com/#System/compmod/system/collections/generic/sorteddictionary.cs

5 BENCHMARKING

Tato kapitola popisuje způsob měření rychlosti různých forem komunikace mezi *managed* a *unmanaged* kódem. Dále jsou prezentovány výsledky tohoto měření v grafické podobě a následně je provedeno jejich vyhodnocení.

Konfigurační nastavení runtime je následující:

- gcServer enabled="true"
- gcConcurrent enabled="true"

Samotný *benchmarking* probíhal na stroji s procesorem *Intel Core i5* rodiny *Ivy Brige*, s 4 GB DDR3 operativní paměti a operačním systémem *Windows 7 Professional 32bit*.

5.1 Sběr dat

Pro sběr byl vytvořen samostatný testovací projekt, který sdružuje a používá ostatní implementace a projekty. Dále byl v rámci experimentu ten samý projekt celý zkopírován a přepracován na používání *blittable* typu *int*. Zároveň byl vytvořen testovací C++ projekt pro otestování C++ dll implementace čistě v nativním prostředí.

5.2 Metodika

Metodika testování je jednoduchá a spočívá v logování počtů procesorových tiků k určitému počtu volání funkce, která vrátí data z neřízené paměti. Do času se samozřejmě započítává i případný marshalling typů.

Testovací data jsou reprezentovaná polem typu *String* a elementy, které jsou generované náhodně a jsou stejné pro všechny testované implementace.

Testovací metoda pomocí parametru typu *bool* rozděluje test na pozitivní nebo negativní. Při pozitivním testu se hodnoty klíčů pro vyhledávání textového řetězce generují náhodně mezi klíčem, který se nachází mezi třemi čtvrtinami z celkového souboru klíčů a posledním klíčem stromu (jinými slovy jde o horní kvartil při sestupném řazení všech dostupných klíčů ve stromě). V opačném případě je to soubor klíčů, které ve stromě neexistují (Obr. 26).

Celkový počet uzlů stromu se rovná jednomu milionu, což není mnoho, ale vzhledem k tomu, že se jedná o přiblížení k reálnému světu, jako příklad je to dostačující.

```
if (negative)
{
    int oneFourthMore = collectionsLenght + (collectionsLenght / 4);
    int overMax = collectionsLenght + 1;
    for (int j = 0; j < callCount; j++)
    {
        keysBuf[j] = random.Next(overMax, oneFourthMore);
    }
}
else
{
    int threeQuarters = collectionsLenght / 4 * 3;
    for (int j = 0; j < callCount; j++)
    {
        keysBuf[j] = random.Next(threeQuarters, collectionsLenght);
    }
}
```

Obr. 26. Generování klíčů pro vyhledávání.

5.3 Implementace

Všechny metody, procedury a funkce byly pojmenované tak, aby na první pohled bylo zřejmé, co daná implementace dělá. Testovací projekt byl řádně zdokumentován. Dokumentace byla tvořena s pomocí nástroje *GhostDoc Community for VS2017*[5].

5.3.1 Logování dat

Pro logování naměřených dat byla vytvořena managed třída *Logger*, která za použití třídy *StreamWriter* zapasuje naměřená data do *txt* souborů ve *csv* formátu. To znamená, že naměřená data nejsou přítomna v paměti dlouhodobě, ale jsou přímo zaevidovaná v textovém souboru. Kvůli použití třídy *StreamWriter* je třída *Logger* opatřena implementací *System.IDisposable.*

Metoda *WriteLine*, která se používá pro zápis dat, je navržena tak, aby byla univerzální. Kontroluje také počet zaslaných dat vůči počtu záhlaví sloupců v textovém souboru. Zároveň tato metoda pracuje s libovolnými objekty, daty, s jejichž pomocí lze metodu *ToString()* konvertovat do textové podoby (Obr. 27).

---

[5] K dispozici zde: https://marketplace.visualstudio.com/items?itemName=sergeb.GhostDoc

```
public static void WriteLine(params object[] values)
{
    Validate(values);
    _WriteValuesToLine(values);
}

private static void Validate(object[] values)...

private static void _WriteValuesToLine(object[] values)
{
    sw.Write(values[0].ToString());
    for (int i = 1; i < values.Length; i++)
    {
        sw.Write(Separator + values[i].ToString());
    }
    sw.Write(sw.NewLine);
    sw.Flush();
}
```

Obr. 27. Metody pro logování dat.

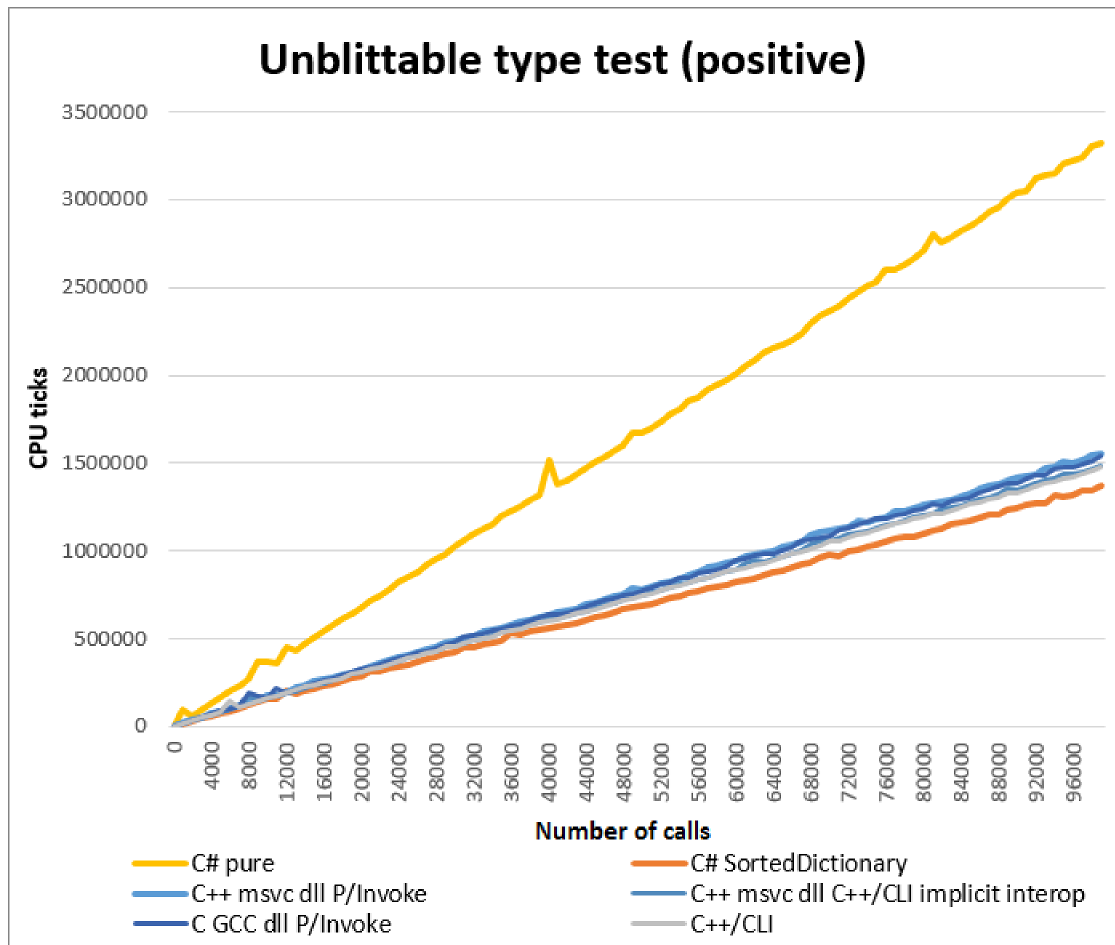
Samotný sběr dat probíhá v jednoduchém algoritmu, který lze vidět na obrázku 28. Celkové měření probíhá v podstatě v cyklu o stu iterací, podle kterých se řadí počet spouštění testovaných metod (viz *callCount),* to znamená že každá metoda v poslední iteraci bude spuštěna až 990 000krát, což je pro zobrazení trendů dostačující.

```
for (int i = 0; i < 100; i++)
{
    callCount = 10000 * i;
    loggerInfo.Add(callCount);
    fillKeysBuf();
    //  C# pure
    Console.WriteLine("C# pure");
    CollectAndPause();
    sw.Start();
    for (int j = 0; j < callCount; j++)
    {
        s = ManagedBTree.FindData(keysBuf[j]).Value;
    }
```

Obr. 28. Úryvek testovacího algoritmu.

5.4 Prezentace výsledků

Výsledky měření jsou prezentované v podobě grafů. Osa *x* reprezentuje počet volání metody. Osa *y* je vždy popsaná a ve většině případů se jedná o procesorové ticky. Implementace binárního stromu, která byla navržena v čistém C#, se v průběhu testů ukázala jako nejpomalejší, a proto není v jiných grafech přítomna (Graf 1).



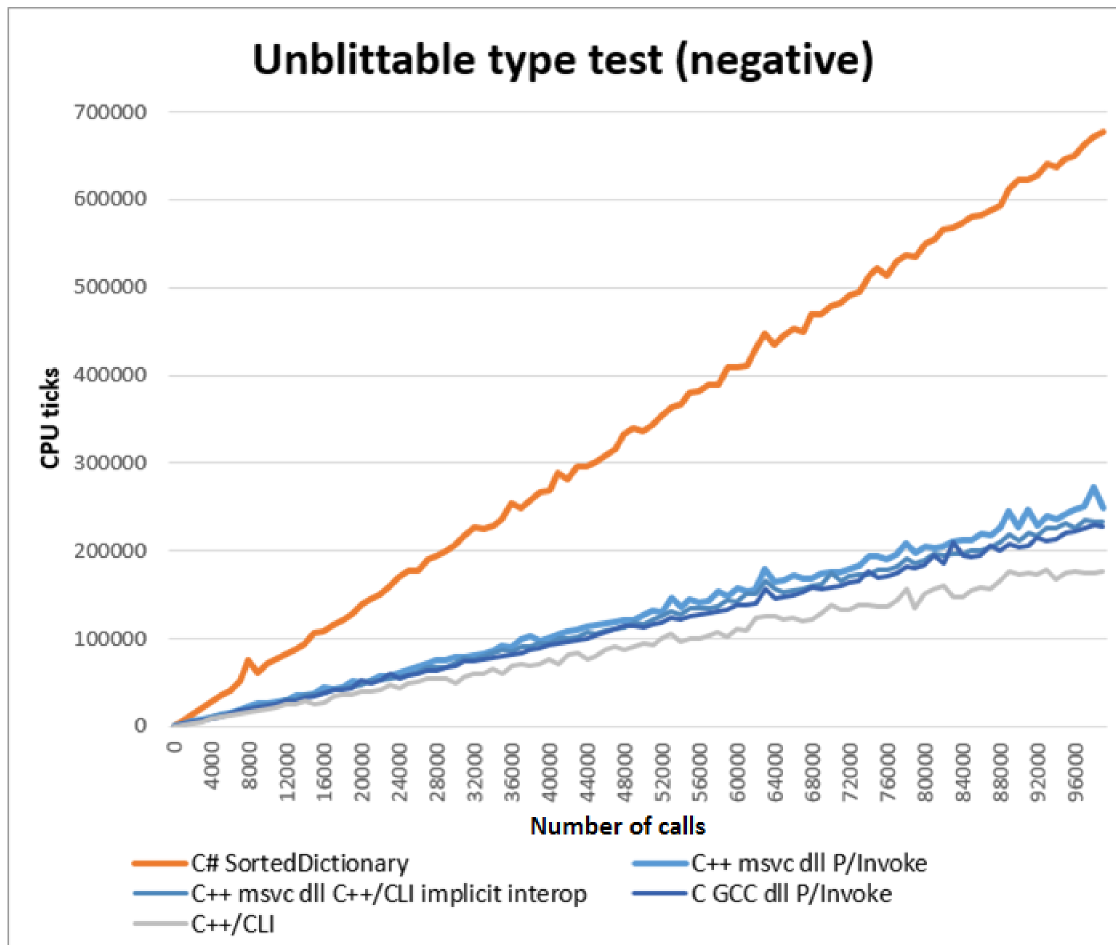
Graf 1. Pozitivní test unblittable typu všech implementací.

5.4.1 Grafy

Na grafu 1 jsou zobrazeny výsledky pozitivního testu, při kterém textový řetězec reprezentovaný *unmanaged unblittable* typem *wchar_t* přešel hranici mezi řízeným a neřízeným kódem, kde byl následně převeden na *managed* typ *String*.

Jak je vidět na zmíněném grafu, řízená implementace *SortedDictionary* je nejrychlejší. *SortedDictionary* je oproti nejrychlejší implementaci s použitím nativního kódu v poslední iteraci rychlejší o cca 7 % a oproti nejpomalejší o cca 13 %. Nejrychlejší nativní verzí je tedy implementace napsaná na C++/CLI s nativním kódem ve stejné *assembly*.

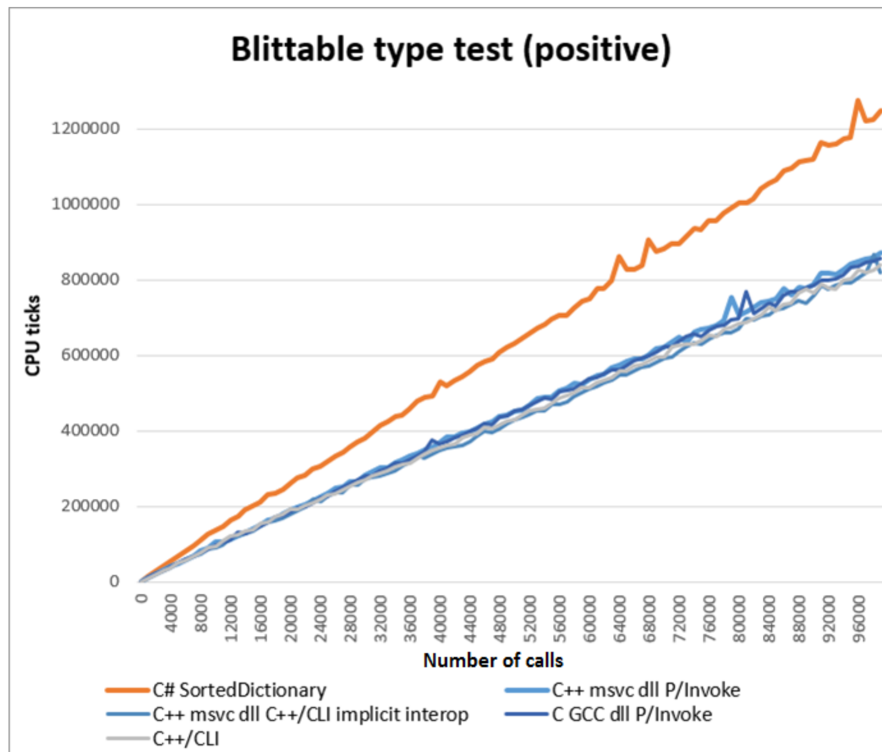
Jak je vidět na grafu č. 2, při negativním průchodu je *SortedDictionary* nejpomalejší. *SortedDictionary* byl v poslední iteraci negativního testu o 380 % pomalejší než nejrychlejší *unmanaged* verze nebo o 250 % než pomalejší nativní implementace. V negativním testu nejrychlejší realizace pracující s unmanaged kódem byla opět verze C++/CLI.



Graf 2. Negativní test unblittable typu.

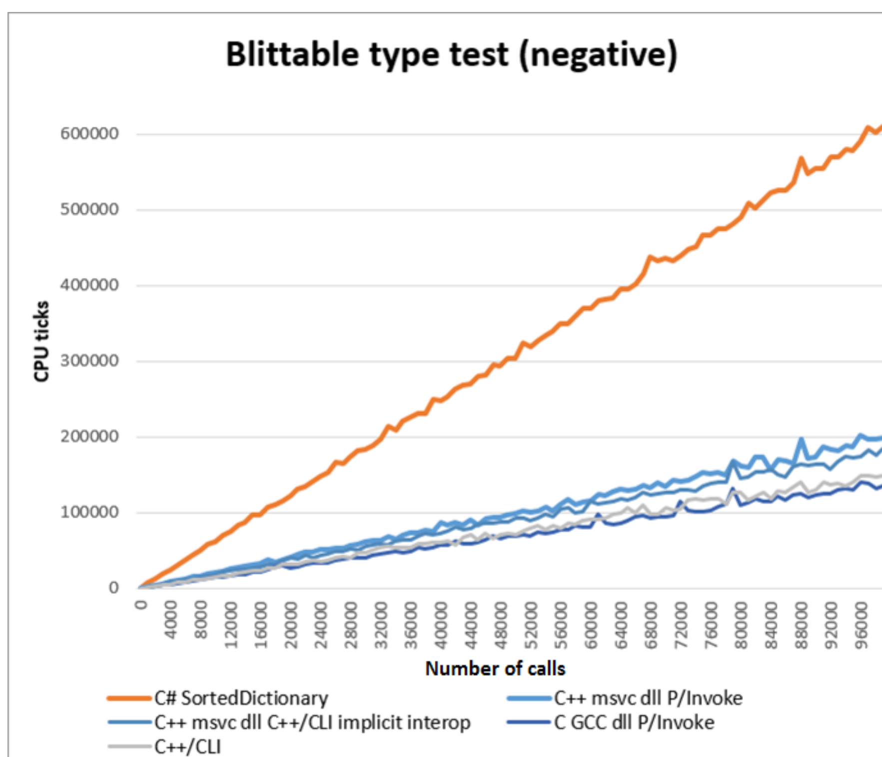
Na grafu 3 jsou zobrazeny výsledky pozitivního testu, při kterém se pro komunikaci mezi řízeným a neřízeným kódem používal *blittable* typ reprezentovaný typem *int*.

Test *blittable* typu ukazuje, že nativní implementace jsou při používání *blittable* typu *int* rychlejší než *managed* implementace *SordedDictionary*. Konkrétně v poslední iteraci je *SordedDictionary* o cca 47 % pomalejší než nejrychlejší *unmanaged* verze nebo o cca 44 % než pomalejší nativní implementace.



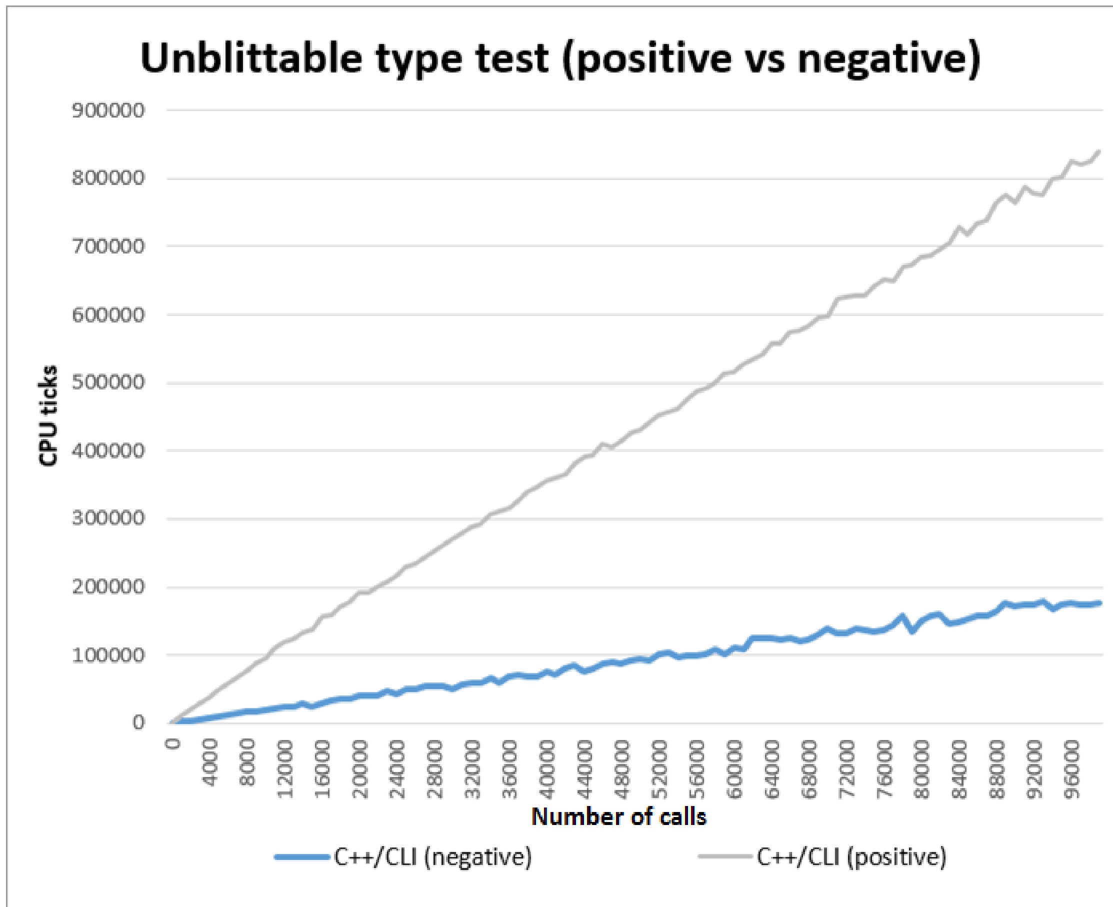
Graf 3. Pozitivní test blittable typu.

Graf 4 pouze potvrzuje již očekávané zpomalení *managed* verze binárního stromu, a to o cca 449 % oproti nejrychlejší nativní nebo o 300 % oproti nejpomalejší.



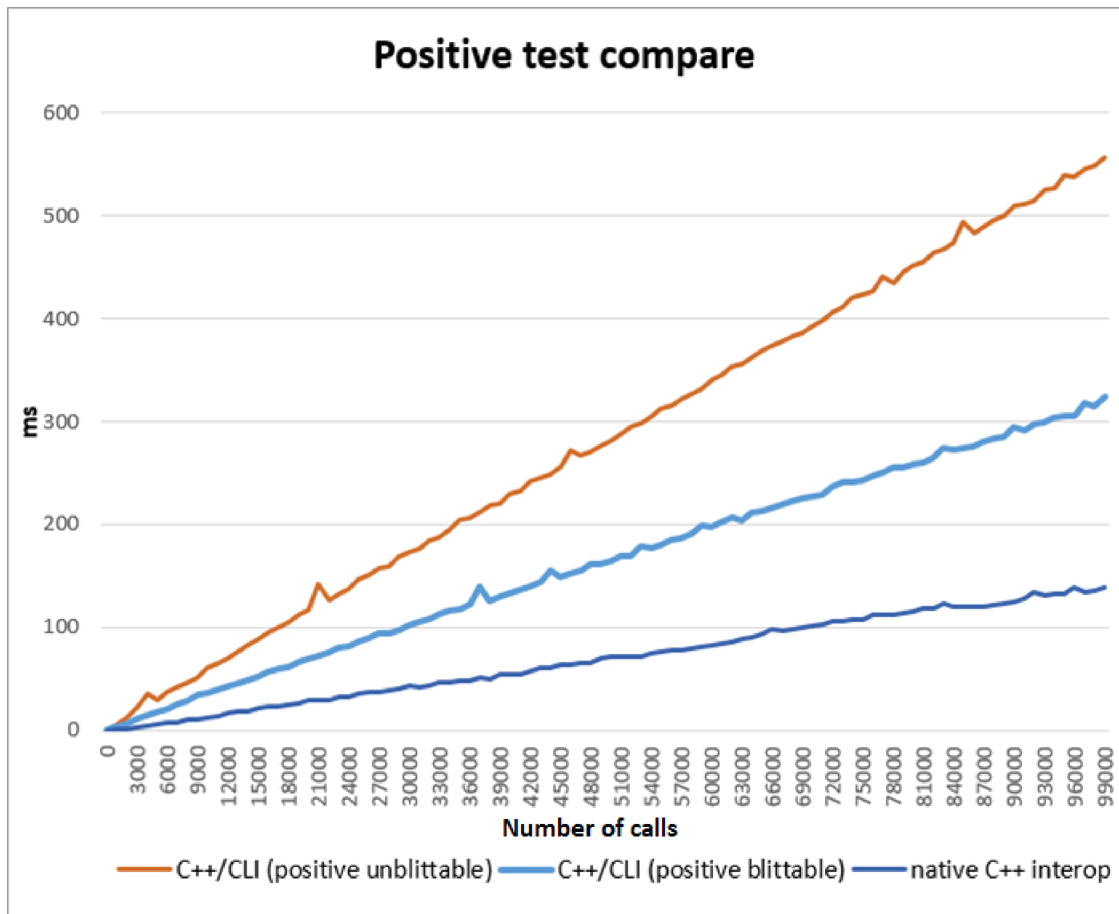
Graf 4. Negativní test blittable typu.

Jelikož nejrychlejší unmanaged implementací ve všech testech byl kód s použitím technologii C++/CLI, graf 5 pro zajímavost zobrazuje rozdíl mezi negativním a pozitivním testem této implementace, který činí v poslední iteraci vysokých 800 %.



Graf 5. Porovnání dvou testů C++/CLI (pozitivní vs negativní).

Graf 6 zobrazuje spojitost mezi nejrychlejší *unmanaged* implementací použitou v *managed* kódu, kterou je C++/CLI. Čistě nativní řešení v jazyce C++ ukazuje skutečnou rychlost prohledání binárního stromu a vracení nalezeného pointeru na textový řetězec typu *wchar*_t. *Blittable.* Řešení C++/CLI je pomalejší o cca 230 % oproti čistě nativní implementaci. *Unblittable* C++/CLI kód je o cca 69 % pomalejší oproti stejnému algoritmu používajícímu *blittable* typ a zároveň o cca 400 % pomalejší než čisté C++ řešení.



Graf 6. Porovnání pozitivních testu C++/CLI a unmanaged C++ interop.

5.5 Vyhodnocení výsledků

Z testů celkově vyplývá, že při pozitivním testu s použitím *unblittable* typu je řízená implementace binárního stromu rychlejší než jakákoli nativní implementace použitá v *managed* prostředí CLR. Naopak při negativním testu s *unblittable* typem hodně zaostává před libovolným nativním řešením. Zároveň je v pozitivním testu dvakrát pomalejší při použití *blittable* typu a překvapivě pomalá při negativním.

Pomalost nativních řešení vysvětluje graf č. 6. Čistě nativní řešení používáme jako konstantu pro vyhledávání odkazu na textový řetězec v neřízené paměti. Jak už bylo uvedeno, při použití *blittable* typů mezi *managed* a *unmanaged* kódem není potřeba takový typ nějak konvertovat, není tedy nutné používat *marshalling* typů. Stále však platí důležitost meziplatformní hranice, jejíž překročení, jak je vidět na grafu, pro tak jednoduchý typ jako *integer* trvá déle (69 %), než kolik času trvalo nativnímu kódu projít celý binární strom. Vysvětluje se tak i obecná pomalost nativního kódu použitého v *managed* prostředí s *unblittable* typy, pro které je nezbytné nutně používat *marshalling* vynásobený délkou textového řetězce.

Při negativních testech je nativní kód rychlejší, protože v případě, kdy klíč nebyl nalezen v neřízené paměti, unmanaged kód vrátí nulový pointer který je typu *DWORD*, což v C# zmámená *blittable* typ *uint* [13]. Zároveň jsou velkým přínosem možnosti C/C++ kompilátorů, které mají mnohem lepší schopnosti v optimalizaci zdrojového kódu než u .NET JIT kompilátoru. V daném výzkumu nebylo bohužel možné kvůli vybranému algoritmu používat vektorovou optimalizaci nebo paralelizaci pomocí kompilátorů v plné míře, ale pokud by to šlo, výsledky by mohly být úplně jiné.

5.5.1 Používání nativního kódu v managed prostředí

Je obecně známo, jaké rychlostní benefity má nativní kód, a právě proto může být používání unmanaged kódu v řízeném prostředí velmi atraktivní. Otázkou však je, kdy se to vyplatí, jak správně takovou komunikaci používat a nakolik je takový způsob udržitelný. V reálném životě by bylo třeba mít dvě optimalizované realizace stejného algoritmu jak pro 32bitovou, tak i pro 64bitovou architekturu a k tomu i dalšího programátora na údržbu nativního kódu.

Jiným případem může být už hotová realizace potřebných algoritmů napsaných na C/C++. Jenomže jak ukazují testy, taková realizace nejlépe funguje ve svém přirozeném nativním prostředí. Platí i zde osvědčené pravidlo, že každý problém má svůj nástroj.

V případě, že nastane potřeba používat nativní kód v řízeném prostředí CLR, lze použít následující doporučení, která jsou výzkumnými zjištěními této práce:

- používat *blittable* typy na všech místech, kde je to možné
- nechávat pro práci s nativním kódem data také v neřízené paměti
- maximálně snížit počet přechodů mezi řízeným a neřízeným kódem
- sdružit přenos dat, tzn. místo četného volání nativní metody poslat data třeba v poli
- implementovat vždy *IDisposable* interface, jestliže zůstávají data mezi voláním metody v neřízené paměti
- používat jednoznačně C++/CLI technologii, která poskytuje lepší ovladatelnost a výkon při interakci s kódem C/C ++
- implementovat při práci s textovými řetězci ruční *marshalling* pomocí *IntPtr*
- vyhýbat se konverzi *Unicode* řetězců do *ANSI*
- vyhýbat se fixování objektu s dlouhým životem.

ZÁVĚR

Tato práce se zabývala sdružením řízeného kódu s neřízeným a následnou analýzou této komunikace z hlediska rychlostního přínosu. Zároveň proběhlo vyhodnocení meziplatformní komunikace a výsledky byly prezentovány v grafické podobě.

Řízené prostředí bylo reprezentované *Common Language Runtime* od společnosti Microsoft a programovacím jazykem C#. Neřízený kód byl zastoupen knihovnami napsanými v jazykách C, C++ a realizací C++/CLI s nativním kódem v této *assembly*. Algoritmus nativního kódu byl reprezentován AVL binárním stromem, což vyloučilo možné paralelní optimalizace C/C++ kompilátorů.

Dále byla prodiskutována problematika používání *unblittable* typů, která pak byla ověřena v praxi na příkladu řízeného typu *String* s několika vlastními realizaci *marshallingu* tohoto typu. Také proběhlo porovnání používání kódu *blittable* s *unblittable* typy a čistě nativního řešení jako protějšku používání *unmanaged* kódu v *managed* prostředí.

Jedním z přínosných výsledků zkoumání bylo testy potvrzené zjištění, že nejlepší možnou technologií pro používání unmanaged kódu v prostředí .NET je technologie C++/CLI. Na základě testu pak byl sestaven doporučující seznam s nejlepšími osvědčenými postupy při používání neřízeného kódu v řízeném prostředí CLR.

SEZNAM POUŽITÉ LITERATURY

[1] MICHAELIS, Mark. *Essential C# 7.0*. 6th Edition. Boston, MA: Addison-Wesley, 2018. ISBN 978-1-5093-0358-8.

[2] RICHTER, Jeffrey. *CLR via C#*. Fourth edition. Redmond, Washington: Microsoft, 2012. ISBN 978-0-7356-6745-7.

[3] TROELSEN, Andrew a Philip JAPIKSE. *Pro C# 7: with .net and .net core*. Eighth Edition. New York, NY: Springer Science+Business Media, 2017. ISBN 978-1-4842-3017-6.

[4] GRIFFITHS, Ian. *Programming C# 5.0*. Beijing: O'Reilly, [2013]. ISBN 978-1-4493-2041-6.

[5] Blittable and Non-Blittable Types. *Microsoft Docs* [online]. 30. 03. 2017 [cit. 2019-05-12]. Dostupné z: https://docs.microsoft.com/cs-cz/dotnet/framework/interop/blittable-and-non-blittable-types.

[6] Interop Marshaling. *Microsoft Docs* [online]. 30. 03. 2017 [cit. 2019-05-12]. Dostupné z: https://docs.microsoft.com/cs-cz/dotnet/framework/interop/interop-marshaling.

[7] Interop with Native Libraries. *Mono* [online]. August 15, 2005 [cit. 2019-05-12]. Dostupné z: https://www.mono-project.com/docs/advanced/pinvoke/.

[8] Platform Invoke (P/Invoke). *Microsoft Docs* [online]. 01/18/2019 [cit. 2019-05-12]. Dostupné z: https://docs.microsoft.com/en-us/dotnet/standard/native-interop/pinvoke.

[9] SIVAKUMAR, Nishant. *C++ / CLI in action*. Greenwich, CT: Manning, c2007. ISBN 1-932394-81-8.

[10] ECMA-372. *C++/CLI Language Specification*. Geneva: ECMA, 2005.

[11] WIRTH, Niklaus. *Algorithms + data structures = programs*. New Jersey: Prentice-Hall, 1976. ISBN 0-13-022418-9.

[12] AHO, Alfred V., John E. HOPCROFT a Jeffrey D. ULLMAN. *Data structures and algorithms*. Reading, Mass.: Addison-Wesley, c1983. ISBN 02-010-0023-7.

[13] Native interoperability best practices: Common Windows data types. *Microsoft Docs* [online]. 18. 01. 2019 [cit. 2019-05-12]. Dostupné z: https://docs.microsoft.com/cs-cz/dotnet/standard/native-interop/best-practices.

SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK

ANSI	American National Standards Institute
API	Application programming interface
AVL tree	Adelson-Velsky and Landis self-balancing binary search tree
B-tree	Binary tree
CIL	Common Intermediate Language
CLI	Common Language Infrastructure
CLR	Common Language Runtime
COM	Component Object Model
CPU	Central processing unit
CSV	Comma-Separated Values
CTS	Common Type System
DLL	Dynamic Link Library
EXE	Executable
GC	Garbage Collector
GCC	GNU Compiler Collection
I/O	Input/Output
IDE	Integrated Development Environment
IL	Intermediate language
Interop	Interoperability
JIT	Just-in-time
JSON	JavaScript Object Notation
MSVC	Microsoft Visual C++
OS	Operační systém
P/Invoke	Platform invoke

PC	Personal computer
RAM	Random-Access-Memory
UTF	Unicode Transformation Format
Win32	Windows 32 bit
XML	eXtensible Markup Language

SEZNAM OBRÁZKŮ

Obr. 1. Metadata a IL kód metody GetText. 12
Obr. 2. Alokace objektů na Managed heap [3, s. 482] 15
Obr. 3. Jednoduchý graf objektů [3, s. 783]. 17
Obr. 4. Sběr odpadků v nulové generaci [3, s. 486] 19
Obr. 5. Finalizační metoda [2, s. 525]. 24
Obr. 6. Mechanismus P/Invoke [9, s. 158]. 25
Obr. 7. Používání atributu DllImport [8]. 26
Obr. 8. Přístup k nativní knihovně pomocí C++ Interop [9, s. 160]. 27
Obr. 9. Unsafe blok [3, s. 438] 27
Obr. 10. Deklarace pointerů v C# [3, s. 440]. 28
Obr. 11. Zobrazení stromových struktur: a) odsazení (indentation); b) vnořené množiny; c) graf; d) vnořené závorky [11, s. 190]. 30
Obr. 12. Dva odlišné binární stromy [11, s. 191]. 31
Obr. 13. Vygenerované atributy pomocí P/Invoke Interop Asistent. 34
Obr. 14. Klicová slova unsafe a fixed. 35
Obr. 15. Nativní procedura Add. 35
Obr. 16. Managed metoda GetText. 36
Obr. 17. Nativni procedura-destructor. 36
Obr. 18. Nativní C++ API. 37
Obr. 19. C++/CLI API. 38
Obr. 20. Nativní kód v C++/CLI assembly. 38
Obr. 21. C++/CLI Marshalling. 39
Obr. 22. C++/CLI destruktor. 39
Obr. 23. C++/CLI mixed mode (implicit C++ Interop). 40
Obr. 24. C++/CLI vytvoření pointeru na textový řetězec. 40
Obr. 25. C# uzel stromu. 41
Obr. 26. Generování klíčů pro vyhledávání. 43
Obr. 27. Metody pro logování dat. 44
Obr. 28. Úryvek testovacího algoritmu. 44

SEZNAM GRAFŮ

Graf 1. Pozitivní test unblittable typu všech implementací. 45
Graf 2. Negativní test unblittable typu...46
Graf 3. Pozitivní test blittable typu. 47
Graf 4. Negativní test blittable typu...47
Graf 5. Porovnání dvou testů C++/CLI (pozitivní vs negativní). 48
Graf 6. Porovnání pozitivních testu C++/CLI a unmanaged C++ interop. 49

SEZNAM TABULEK

Tab. 1. Blittable typy C# a blittable nativní typy [7]. 23

SEZNAM PŘÍLOH

P1 DVD s diplomovou práci, obsahující projektové soubory a zdrojové kódy