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

MINISTERSTVO ŠKOLSTVÍ, MLÁDEŽE A TĚLOVÝCHOVY

Název projektu	Rozvoj vzdělávání na Slezské univerzitě v Opavě
Registrační číslo projektu	CZ.02.2.69/0.0./0.0/16_015/0002400

Úvod do programování

Distanční studijní text

Radomír Perzina

Karviná 2021

SLEZSKÁ UNIVERZITA
OBCHODNĚ PODNIKATELSKÁ FAKULTA V KARVINÉ

Obor: Informatika

Klíčová slova: Programování, C#, Microsoft Visual Studio, ladění, proměnná, datový typ, operace, operátor, metoda, podmínka, cyklus, pole, parametr, soubor, algoritmus, rekurze.

Anotace: Tento text je určen studentům bakalářského stupně studia na Slezské univerzitě, Obchodně podnikatelské fakultě v Karviné. Jakožto studijní opora je určen především studentům kombinované formy studia, mohou jej však stejně dobře využít i studenti prezenční formy. Tato opora je zaměřena na základy zejména strukturovaného programování, proto ji nejvíce ocení studenti, kteří s programováním teprve začínají, ale některé informace budou užitečné i pro mírně pokročilé. Jsou zde vysvětleny základní konstrukce a příkazy jazyka C#. Text je orientován na praktické použití, proto jsou všechny příkazy a konstrukce demonstrovány na ukázkách reálného kódu v Microsoft Visual Studiu. Výukový text je členěn tak, že v první kapitola je věnována historii programování a seznámení s vývojovým prostředím. Následujících pět kapitol se podrobněji věnuje základním konstrukcím a příkazům. Poslední kapitola potom demonstruje využití nabytých znalostí pro implementaci některých prakticky využitelných algoritmů.

Autor: **Ing. Radomír Perzina, Ph.D.**

Obsah

ÚVODEM

Do rukou se Vám dostává výukový text, který se snaží čtenáře seznámit se základy programování v jazyce C# a se základy používání vývojového prostředí Microsoft Visual C#. Tento text je vhodný jako distanční opora ve výuce předmětů na vysokých školách se zaměřením na informatiku. Obsahem výkladu je stručné seznámení s historií programování, výklad základních příkazů a konstrukcí jazyka C# a ukázka několika jednoduchých algoritmů využitelných v praxi.

Text je strukturován do sedmi kapitol. Každá kapitola začíná stručným seznámením s jejím obsahem v rychlém náhledu kapitoly, dále obsahuje stručné cíle a klíčová slova. V samotném textu se vyskytují distanční prvky, které čtenáře upozorní na důležité části k zapamatování a na texty pro zájemce. Každá kapitola končí kontrolními otázkami, krátkým shrnutím a odpověďmi na otázky.

RYCHLÝ NÁHLED STUDIJNÍ OPORY

V současné době jsou základy programování důležitou součástí každého absolventa vysoké školy nejen se zaměřením na informatiku. Cílem této opory je poskytnout čtenáři základní seznámení s jazykem C# a základy algoritmizace. Text je členěn do sedmi kapitol.

V první kapitole je zmíněna stručná historie programovacích jazyků a jsou zde demonstrovány základy práce s vývojovým prostředím Microsoft Visual Studio. Druhá kapitola je zaměřena na uchovávání dat různých datových typů pomocí proměnných, součástí je i uvedení základních operací pro práci s čísly. Třetí kapitole se věnuje způsobu, jak můžeme kód rozčlenit do menších úseků pomocí metod. Cílem čtvrté kapitoly je uvést možnosti podmínečného vykonání určitých částí kódu, důležitou částí jsou i logické výrazy a operátory. Pátá kapitole se zabývá cykly a poli. V šesté kapitoly jsou uvedeny základní možnosti pro komunikaci programu s okolím. Závěrečná kapitola potom aplikuje nabyté znalosti z předchozích kapitol na implementaci jednoduchých algoritmů využitelných v praxi.

1 ZÁKLADY PROGRAMOVÁNÍ

RYCHLÝ NÁHLED KAPITOLY

V této kapitole se dozvíte, jaký software budeme v rámci studia programování pomocí této knihy používat, jak jej získat, nainstalovat a jak s ním pracovat. Dozvíte se něco historii programování, co je to program, programovací jazyk a vytvoříte si svůj první program v jazyce C#. Neopomeneme ani komentáře a jakým způsobem můžeme hledat a odstraňovat chyby v programu.

CÍLE KAPITOLY

Po prostudování této kapitoly budete umět:

- Vyjmenovat z historického hlediska nejvýznamnější programovací jazyky.
- Nainstalovat Microsoft Visual Studio.
- Vytvořit jednoduchý program v jazyce C#.
- Okomentovat části kódu.
- Definovat rozdíl mezi syntaktickou a sémantickou chybou.
- Krokovat program.
- Vložit do programu bod přerušení.

KLÍČOVÁ SLOVA KAPITOLY

Program, programování, MS Visual studio, komentář, syntaktická chyba, sémantická chyba, ladění, bod přerušení.

1.1 Historie programovacích jazyků

Historie informatiky nebo, chcete-li, výpočetní techniky začíná už v době starověkých civilizací výpočetními pomůckami (abakus, čínské počítací hůlky) a pokračuje v novověku od dob renesance dokonalejšími mechanickými kalkulátory buď digitálními (konstrukce Shickardova, Pascalova, Leibnizova, Colmarova, Odhnerova), nebo analogovými (logarit-

mické pravítko), aby v 19. století mechanická etapa vývoje vyvrcholila velkolepými projekty jednoúčelových počítačů – diferenčních strojů, jež dokázaly automaticky počítat a tisknout tabulky matematických funkcí, a Hollerithových děrnoštítkových systémů, jež dokázaly zpracovávat rozsáhlé relační databáze.

První počítače měly velmi daleko k dnešním univerzálním nástrojům, provádějícím výpočty, zpracovávajícím informace různých druhů a umožňujícím rychlou komunikaci prostřednictvím světové počítačové sítě internet. První počítače prováděly skutečně „pouze“ výpočty, které předtím ručně realizovaly týmy lidských počtářů. V angličtině se právě v tuto dobu posunul význam slova computer od původního počtář (člověk profesionálně provádějící výpočty) k počítač (stroj, který provádí výpočty).

První generace počítačů se programovala ručním propojováním vodičů (propojováním zdířek spojovacími vodiči na ovládacím panelu, přepínáním přepínačů apod.), k němuž s o něco později přidalo čtení vstupních instrukcí a dat z papírové děrné pásky. Přeprogramování počítače na jinou úlohu byl pracný úkon, který vyžadoval trpělivost a pečlivost a souhru několika členů týmu. Kromě programátorů, kteří vymysleli a zapsali algoritmus, byli potřeba kodéři, kteří přepsali celý program do posloupnosti dvojkových čísel a operátoři, kteří řídili vlastní spuštění a provedení kódu. Ladění programu bylo velmi obtížné a časově náročné, protože pro něj neexistovaly žádné pomůcky.

Další nevýhodou první generace byla neexistence jednotících norem. Co počítač, to originál s jinou sadou příkazů procesoru, s jiným strojovým kódem. Programátor musel myslet nejen na algoritmus, ale i na uložení kódu v operační paměti, stejně jako na alokaci paměti pro všechny proměnné, pole proměnných atd. To vše v absolutní pozici v paměti, tedy konkrétní instrukce programu (početní operace, rozhodovací operace, skok v programu, …) měla v podstatě předem pevně dané místo operační paměti, ve kterém byla uložena.

S rozvojem výpočetní techniky bylo brzy zřejmé, že software (strojový kód) zaostává za možnostmi hardware a že je potřeba vymyslet novou „technologii“ programování, která by využití počítačů nebrzdila. Prvním krokem od strojového kódu k symbolickému jazyku přístupnému člověku byla skupina jazyků symbolických adres (JSA; angl. assembly language), které místo číselných kódů pro jednotlivé instrukce procesoru používaly symbolické zkratky (např. ADD pro součet, JMP pro skok, MOV pro přesun hodnoty z paměti do registru) a současně nepožadovaly absolutní alokaci paměti, místo níž použily symbolické adresy, jejichž konkretizaci pak zařídil překladač typu assembler (tedy „sestavovač“, program, který z programu v JSA sestaví strojový kód). Často se nesprávně název assembler používá i pro JSA, ve skutečnosti je assembler historicky první typem překladače programovacího jazyka, který překlad z JSA do strojového kódu provádí na několik průchodů, při nichž postupně provede potřebnou alokaci paměti pro vlastní program i pro potřebné proměnné.

Assemblery se pro některé speciální úlohy používají i v současnosti. Např. tam, kde je nutný přímý přístup k hardware (ovladače zařízení), vysoký výpočetní výkon, nebo řízení procesů v reálném čase. V JSA se také psaly první operační systémy pro sálové počítače.

Dalším typickým prvkem, který usnadní programování v JSA, je zavedení pseudoinstrukcí a maker, čímž se zjednoduší zápis opakujících se částí kódu. Jde o předstupeň plnohodnotných podprogramů.

JSA jsou tak jako strojový kód vázané na konkrétní typ, resp. konkrétní řadu procesorů. Programy napsané v JSA pro určitý hardware nejsou buď přenositelné vůbec, nebo přenositelné jen s velkými obtížemi na jinou hardwarovou platformu. JSA přinesly programátorům velké zjednodušení práce, byly typickým nástrojem druhé generace počítačů, ale bylo zřejmé, že ani ony nejsou elegantním a jednoduchým řešením zápisu algoritmů. Na cestě od stroje k člověku bylo potřeba učinit další krok.

Třetí generaci softwaru představují obecné problémově orientované programovací jazyky. Často se jim říká také vyšší programovací jazyky a myslí se tím, že se od nízké úrovně strojového kódu či JSA dostali značně vysoko směrem k algoritmickému myšlení člověka plánujícího složitý výpočet.

Podobně označení „problémově orientovaný“ znamená, že programátor nemusí myslet na procesor a jeho instrukční sadu, ale může se plně soustředit na problém, který chce vyřešit. V praxi to značí nezávislost na konkrétním hardware, a tedy snadnou přenositelnost napsaných programů.

Takových programovacích jazyků existovala už na konci 50. let celá řada. Kdybychom se snažili postihnout celou šíři vývoje, hrozilo by, že ztratíme potřebný nadhled. Proto se soustředíme pouze na tři nejvýznamnější představitele vývoje, za něž považuji jazyky FORTRAN, COBOL a ALGOL.

Jazyk FORTRAN (z angl. FORmula TRANslator) vyvíjela ve firmě IBM od roku 1954 skupina, kterou vedl John Backus (1924 – 2007). Důraz při vývoji jazyka a překladače byl kladen na rychlost a efektivnost zejména vědecko-technických výpočtů, přenositelnost nebyl na prvním místě, protože šlo o projekt orientovaný na sálové počítače IBM řady 700. Díky této koncepci se FORTRAN stal jedním z nejrozšířenějších programovacích jazyků své doby. Vývoj jazyka intenzivně pokračoval do roku 1963 (FORTRAN IV), jazyk byl později normalizován (standardy ANSI a ISO), byly vydávány jeho nové verze a své uživatele má ještě dnes (FORTRAN 2008).

John Backus je znám také jako spolutvůrce Backus-Naurovy formy pro popis gramatik formálních jazyků. Používá se jak pro popis gramatik programovacích jazyků, instrukčních sad, komunikačních protokolů a dokonce i částí gramatik skutečných jazyků. Ještě před jazykem FORTRAN vyvinul jazyk Speedcoding, který se však nerozšířil. Později spolupracoval na vývoji jazyka ALGOL a také podnítil zájem o funkcionální programování, o němž budeme mluvit později.

Jazyk COBOL (z angl. COmmon Business Oriented Language) byl oficiálně vydán roku 1959 sdružením CODASYL, založeným z iniciativy Ministerstva obrany USA. Administrativní pokyn vlády USA z roku 1960 pak nařídil vybavit překladačem COBOLu každý

komerčně instalovaný počítač, čímž došlo jeho k masovému rozšíření. Matkou COBOLu bývá nazývána Grace Hopper (1906 – 1992), která stavěla na předchozích zkušenostech z práce matematičky a programátorky ve firmě Eckert-Mauchly Corporation (výrobce počítačů BINAC a UNIVAC), pro níž a pro firmu Remington Rand zpracovala programovací jazyky FLOW-MATIC a MATH-MATIC. Tato dáma je autorkou první příručky počítačové terminologie a matkou myšlenky použití jednotné matematické notace a jednoduchých anglických příkazů k sestavení zápisu programu v podobě srozumitelné člověku i počítači. COBOL umožnil kromě vědecko-technických výpočtů také základní zpracování databází. Po formální stránce nedosáhl přehlednosti FORTRANu či ALGOLu, ale z praktického hlediska šlo o snadno implementovatelný a dobře použitelný programovací jazyk pro běžné rutinní výpočty v komerční sféře.

Oba výše uvedené jazyky vznikly v USA a odtud se šířily do celého světa. Třetí – programovací jazyk ALGOL (z angl. ALGOrithmic Language) vznikl z iniciativy evropského Odborného výboru pro programování (GAMM), která na společném zasedání s americkou ACM (Association for Computing Machinery) v roce 1958 vydala první verzi jazyka – ALOGOL 58. Velmi rychle byla zapracována řada připomínek a vylepšení a vydána verze ALGOL 60. Významný podíl na vývoji konečné verze ALGOL 68 měl svými pracemi, věnovanými operativní sémantice, holandský informatik Adriaan van Wijngaarden (1916 – 1987). Jazyk ALGOL se od konce 60. let dále nevyvíjel, nikdy nedosáhl tak masového rozšíření v praxi jako FORTRAN či COBOL, ale se stal obecně uznávaným standardem pro publikování algoritmů v odborných časopisech a zároveň byl východiskem a inspirací pro řadu dalších programovacích jazyků, např. Pascal, Simula 67 či Ada.

V roce 1961 přinesla titulní strana časopisu „Communications of the ACM“ (Vol. 4, No. 1) seznam 73 různých programovacích jazyků a v průběhu 60. let tento počet dál prudce rostl. Proto se budeme věnovat opět jen několika vybraným jazykům. Hlavním hlediskem pro výběr je rozšířenost jazyků v programátorské praxi.

Velmi populárním jazykem je pro svoji jednoduchost BASIC (Bigginer’s All-purpose Symbolic Instruction Code), který roku 1963 navrhli John Kemeny (1926 – 1992) a Thomas Kurtz (*1928) a který zažil obrovský rozmach v 80. letech jako základní programovací nástroj na osmibitových domácích počítačích a v současnosti zažívá comeback díky firmě Microsoft a jejím produktům Visual Basic, MS Office (makrojazyk VBA) a MS Internet Explorer (skriptovací jazyk VB Script).

Společnou nevýhodou FORTRANu a BASICu je fakt, že nenutí programátory k přehledným, dobře strukturovaným zápisům. Proto se i sám tvůrce programu může s určitým časovým odstupem špatně orientovat ve zdrojovém kódu, který vypracoval. To spolu s otázkou přesného popisu struktury dat, která pak usnadní psaní algoritmu, motivovalo profesora Niklause Wirtha (*1934) z Vysoké školy technické v Curychu k vytvoření programovacího jazyka Pascal. První implementaci Pascalu realizoval Niklaus Wirth v roce 1970.

Popularizaci jazyka prospěla levná a přitom kvalitní implementace Turbo Pascal, díky které se masově rozšířil na osobních počítačích typu IBM PC. Ta vznikla ve firmě Borland

v roce 1983 a hlavní zásluhy na jejím vývoji mají dánský programátor Anders Hejlsberg (*1960) a francouzský informatik Philippe Kahn (*1962). Jazyk Pascal je dodnes součástí vývojového prostředí Delphi. Kromě toho existují open source alternativy Free Pascal a Lazarus. Niklaus Wirth navrhl několik dalších programovacích jazyků (Algol W, Modula, Oberon) a je autorem vynikající, dodnes používané učebnice programování Algorithms + Data Structures = Programs.

Dalším významným jazykem začátku 70. let byl v roce 1972 programovací jazyk C. Jeho autorem je Denis Ritchie (*1941). Referenční příručku „The C Programming Language“ napsal v roce 1978 společně s Brianem Kernighanem (*1941). Popsaná verze se stala de facto standardem K&R jazyka C.

Na přelomu 60. a 70. let se objevují také první objektově orientované jazyky. Nejprve jazyk Simula norských informatiků Ole-Johana Dahla (1931 – 2002) a Kristena Nigaarda (1926 – 2002), který sice nebyl masově rozšířený, ale svým přístupem k řešení problémů položil základy objektově orientovaného programování. Na práce norských průkopníků pak navázal americký počítačový vědec Alan Key (*1940) s jazykem Smalltalk, který svou důslednou objektovostí byl mimo jiné vhodný k programování grafických aplikací. Díky tomu se Allan Key stal také tvůrcem grafického uživatelského rozhraní se systémem překrývajících se oken, jaké známe z moderních operačních systémů pro osobní počítače. Zajímavé je, že Alana Keye kromě prací norských předchůdců ovlivnil také Seymour Papert (autor jazyka LOGO), který ho přivedl ke studiu konstruktivismu.

Významným procedurálním jazykem je z historického hlediska rovněž Forth. Vyvinul jej roku 1970, nejprve pro svoji osobní potřebu Charles Moore (*1938) a použil jej pro ovládání soustavy radioteleskopů v Národní radioastronomické observatoři (NRAO) na hoře Kitt Peak v Arizoně. O jazyk však měli zájem další programátoři a rychle se rozšířil, i když ne tak masově jako výše uvedené jazyky. Roku 1973 založil Charles Moore společnost Forth inc. a začal jazyk nabízet komerčně. Roku 1976 se jazyk stal standardem programování v Mezinárodní astronomické unii. Dodnes je udržován a dále rozvíjen. Forth je typický prací se zásobníky a použitím postfixové notace. Nepotřebuje pro svoji práci operační systém a hodí se pro programování řídících a měřicích systémů, založených na mikroprocesorech s kratší délkou slova (dnes 8, 16 bit), tedy ve vestavěných (angl. embedded) systémech a pro práci v reálném čase.

Již jsme se zmínili, že John Backus (1924 – 2007) podnítil zájem o funkcionální programování. Přestože byl autorem komerčně velmi úspěšného procedurálního jazyka FORTRAN, nepovažoval procedurální přístup za jediný možný. Při převzetí Turingovy ceny v roce 1977 přednesl přednášku s názvem Může být programování osvobozeno od von Neumannova stylu?, v níž představil projekt jazyka FP. Tento projekt byl dokončen a distribuován, na rozdíl od pozdějšího FL (Function Level), který zůstal na úrovni firemního projektu IBM a distribuován nebyl. Podnítil však zájem o směr deklarativního programování, kterému říkáme funkcionální.

Moderním funkcionálním jazykem je Miranda, navržená v roce 1985 britským informatikem Davidem Turnerem (*19..) a její následník Haskell vyvinutý v roce 1990 skupinou informatiků a dále zdokonalovaný v současnosti. Jazyk Haskell je pojmenován po matematikovi, který vybudoval matematické základy funkcionálního programování. Byl to Haskell Brooks Curry (1900 – 1982), zakladatel kombinatorické logiky.

Mnohem starším funkcionálním jazykem je LISP (LISt Processing), který pochází z roku 1960. Jeho autorem je John McCarthy (1927), který je známý také svým výrazným podílem na vymezení pojmu umělá inteligence. V oblasti umělé inteligence je LISP stále využíván. Má jednoduchou strukturu, protože nerozlišuje mezi kódem a daty. Na vše se dívá jako na seznamy. Problematická je nutnost používání velkého množství závorek. Staví na něm některé systémy pro zpracování jazyků, ať přirozených (editor textů Emacs), či formálních (systém počítačové algebry Maxima). Vychází z něj další programovací jazyky (Smalltalk, Scheme) a dokonce i dětský programovací jazyk Logo.

Roku 1972 navrhl francouzský informatik Alain Colmerauer (*1941) deklarativní, neprocedurální jazyk Prolog, založený na predikátové logice prvního řádu. Deklarativní jazyk je přímým opakem procedurálního, to znamená, že popíšeme výchozí situaci (zadání) a cíl výpočtu, ale vůbec neříkáme, jakým postupem by se program měl k výsledku dostat. Počítač vlastně výsledek vyvozuje ze zadání pomocí pravidel formální matematické logiky. Proto mluvíme o logickém programování.

Novějším logickým programovacím jazykem je jazyk Gödel, který roku 1992 vytvořila dvojice informatiků John Lloyd a Patricia Hillová. Byl nazván na počest rakouského logika, matematika a filozofa, brněnského rodáka Kurta Gödla (1906 – 1978).

1.2 Jazyk C#

Jazyk C# je vysokoúrovňový objektově orientovaný programovací jazyk vyvinutý v roce 2002 firmou Microsoft zároveň s platformou .NET Framework, později schválený standardizačními komisemi ECMA (ECMA-334) a ISO (ISO/IEC 23270). Jazyk C# je založen na jazycích C++ a Java a je tedy nepřímým potomkem jazyka C, ze kterého čerpá syntaxi.

Název jazyka C# (vyslovované anglicky jako C Sharp, /si: ša:p/) je odvozen z hudební notace, kde křížek označuje zvýšení noty o půl tónu a v tomto případě by označoval notu cis, tedy C zvýšené o půl tónu. Podobně vznikl název jazyka C++ jako zlepšení jazyka C: „++“ totiž v syntaxi jazyka C znamená zvýšení hodnoty proměnné o 1. Křížek na počítačové klávesnici (#) a křížek v hudební nauce (♯) jsou dva odlišné znaky. Pro zápis názvu jazyka C Sharp se nepoužívá znak hudebního křížku z technických důvodů, protože tento se na standardní klávesnici nevyskytuje, ale pro zjednodušení se používá klasický křížek. Toto je zakotveno ve specifikaci jazyka C#, ECMA-334. Toto opatření je spíše praktického rázu, takže v případech jako jsou různé marketingové materiály se stále často používá znak křížku z hudební notace.

Od doby svého vzniku se stal jazyk C# velmi populární a dnes ho můžeme používat za standard pro vývoj různých druhů aplikací, jako např. databázových aplikací, webových aplikací i služeb, formulářových aplikací i aplikací pro mobilní zařízení.

Jazyk C# podobně jako jazyk Java využívá pro svoji činnost takzvaný virtuální stroj. Zdrojový kód v jazyce C# je nejprve přeložen do tzv. mezikódu, kterému říkáme CIL (Common Intermediate Language). Jedná se v podstatě o strojový (binární) kód, který má ale o poznání jednodušší instrukční sadu a přímo podporuje objektové programování. Tento mezikód je potom díky jednoduchosti relativně rychle interpretovatelný tzv. virtuálním strojem (tedy interpretem, v případě .NET je to tzv. CLR - Common Language Runtime). Výsledkem je strojový kód pro náš procesor.

Ve spojitosti s programovacím jazykem C# se často setkáme s pojmem .NET Framework. .NET Framework je softwarová platforma poskytující širokou škálu prostředků pro programy. Její popis by zcela jistě vydal na celou řadu knih a není cílem této publikace ji detailně popisovat. Prostředky z této platformy budeme při programování v jazyce C# používat, např. pro výpis na konzoli.

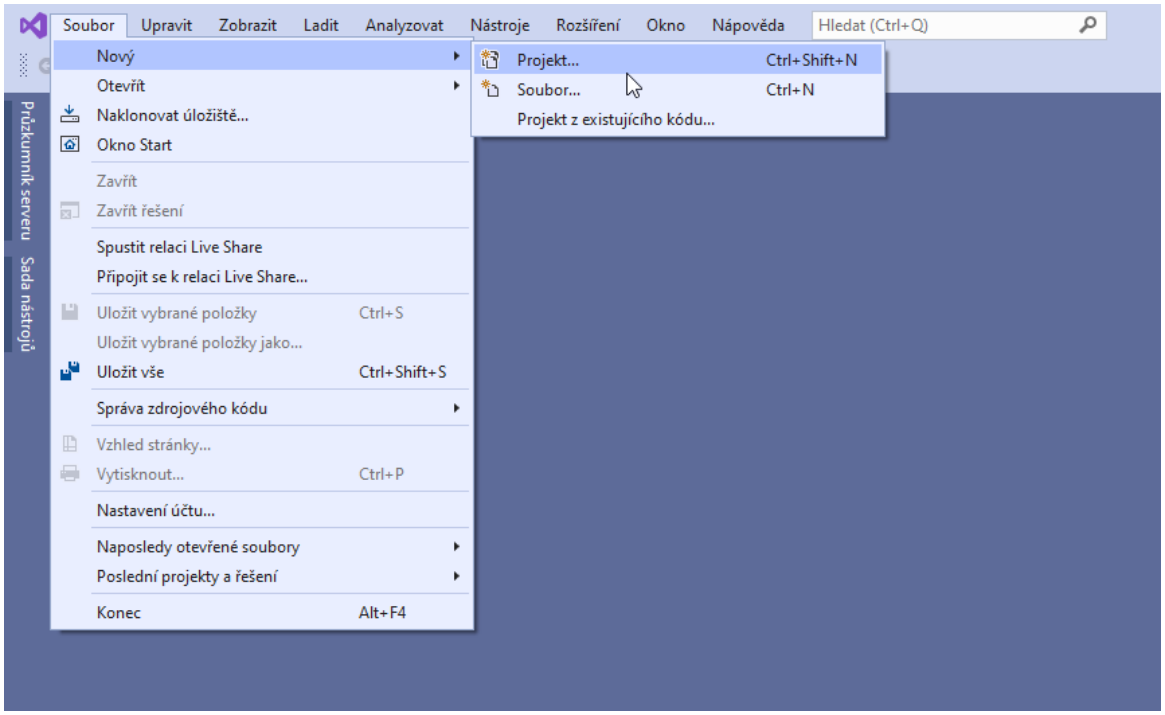
1.3 Základy práce s Microsoft Visual Studiem

1.3.1 Instalace

Než začneme vytvářet náš první program je nezbytné si nainstalovat vývojové prostředí Microsoft Visual Studio. V tomto textu budeme používat verzi Microsoft Visual Studio 2019 Community Edition, která je k dispozici zdarma ke stažení zde: https://visualstudio.microsoft.com/vs/community/. Vývojové prostředí nainstalujte podle pokynů uvedených na webových stránkách a v instalačním průvodci vývojového prostředí Microsoft Visual Studio.

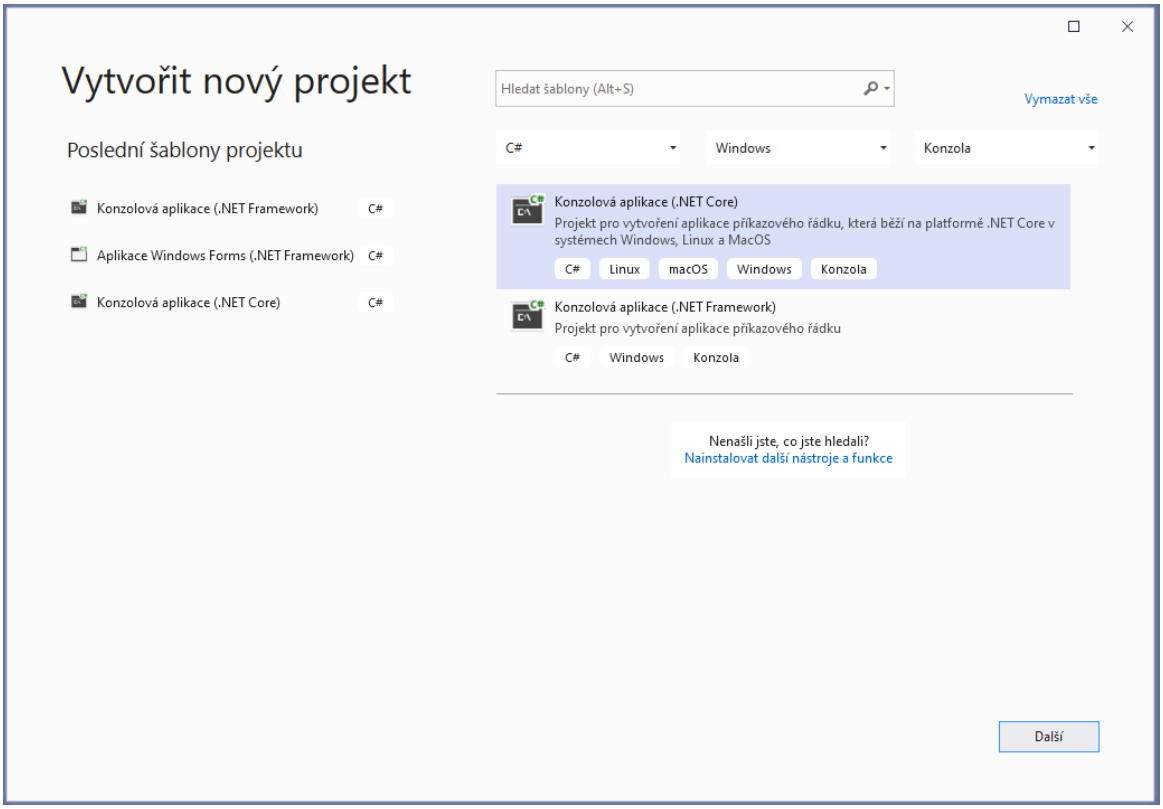
1.3.2 První program

Pokud máte nainstalované vývojové prostředí Microsoft Visual Studio, se můžete pustit do vytvoření svého prvního programu v jazyce C#. Spusťte vývojové prostředí Microsoft Visual Studio. Zobrazí se vám úvodní obrazovka, na které vyberte z hlavního menu možnost *Soubor* → *Nový* → *Projekt*.



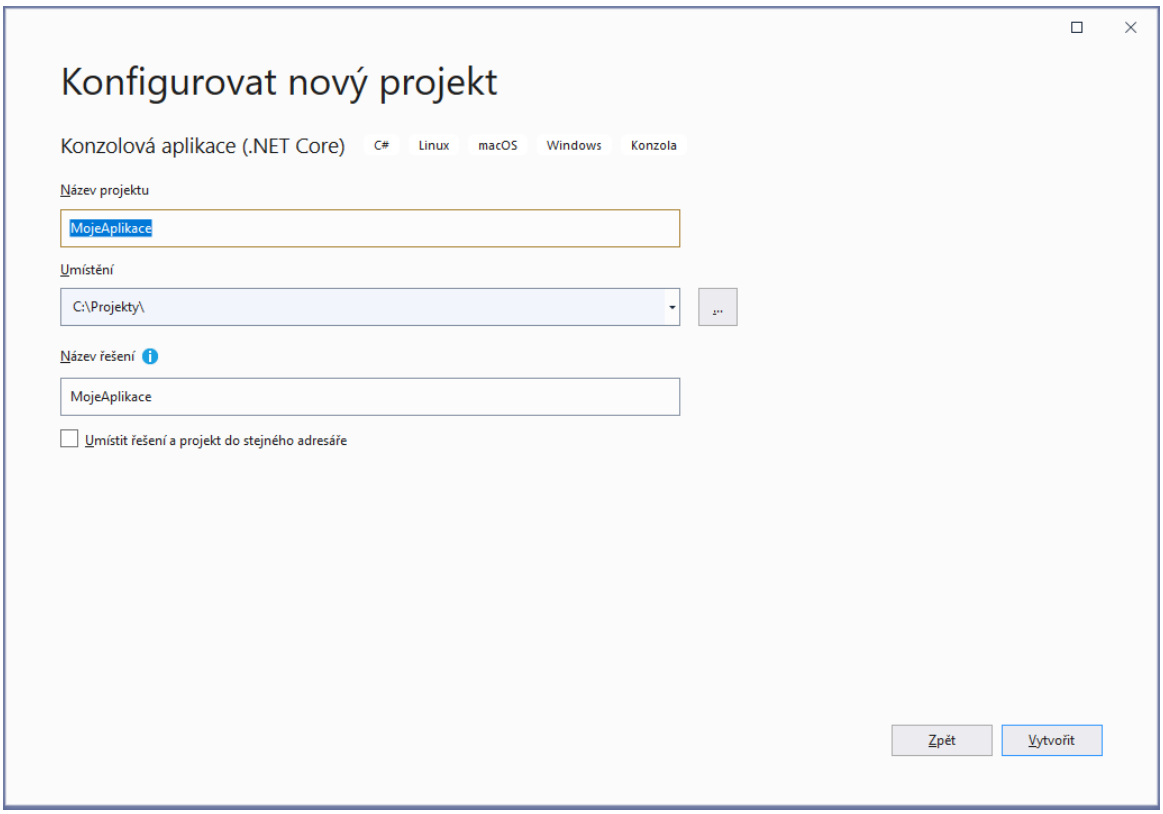
Obrázek 1.1: Nový projekt

Zobrazí se nám poměrně široká nabídka typu projektů, které můžeme filtrovat podle tří kritérií. V prvním kritériu si vybereme programovací jazyk, v němž chceme naši aplikaci vytvořit – zde zvolíme *C#*. Druhé kritérium určuje cílový operační systém – zvolme *Windows* a poslední položka je upřesnění typu aplikace, kde si vybereme položku *Konzola*. Po zvolení těchto kritérií se nám zobrazí dva typy projektů Konzolová aplikace (.NET Core) a Konzolová aplikace (.NET Framework). Pro naše účely si můžeme zvolit kteroukoliv z těchto dvou možností, přičemž .NET Core je novější, proto i my zvolíme tuto možnost.



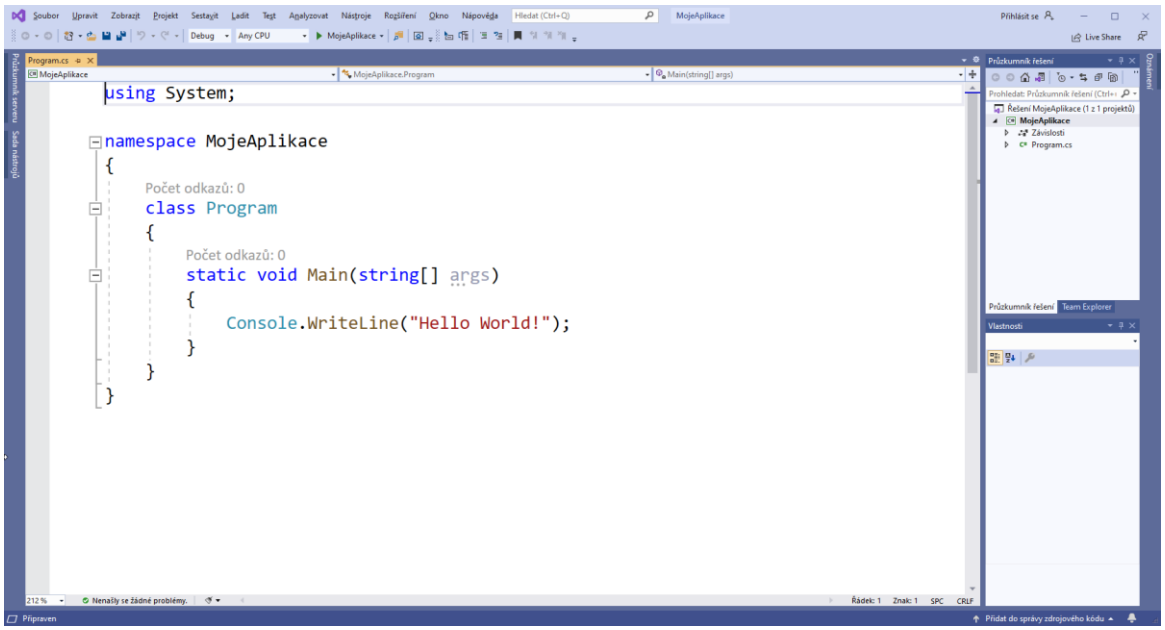
Obrázek 1.2: Typ projektu

V dalším kroku musíme vyplnit název našeho projektu a jeho umístění na disku.



Obrázek 1.3: Název projektu

Po kliknutí na tlačítko *Vytvořit* se nám vytvoří nový projekt. Dříve než se pustíme do vlastního programování řekneme si něco o rozložení oken v tomto prostředí. Vpravo nahoře se nachází okno Průzkumník řešení, které obsahuje seznam souborů, z nichž se projekt skládá. Vpravo dole je potom okno Vlastnosti, které zobrazuje vlastnosti aktuálně vybraného prvku v jiných oknech, např. úplnou cestu k vybranému souboru. Poslední okno, která zabírá největší plochu obrazovky, obsahuje zdrojový kód aktuálně vybraného souboru.

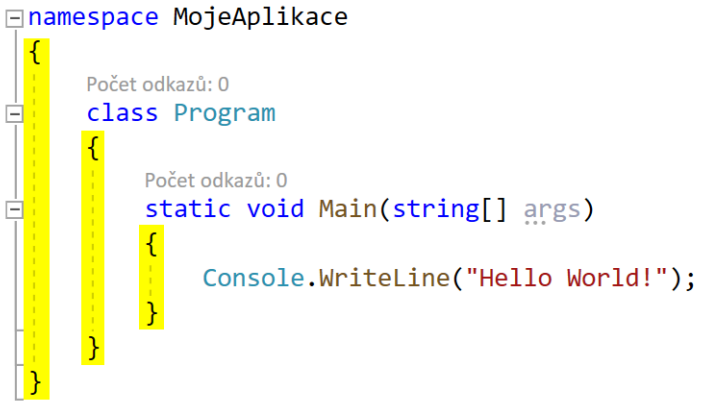


Obrázek 1.4: Prázdný projekt

Můžeme si všimnout, že zdrojový kód není úplně prázdný, ale obsahuje už předpřipravenou šablonu, která obsahuje základní strukturu programu, čímž je nám ušetřena práce psaním našeho programu úplně od píky.

Nyní si projděme základní strukturu programu. První řádek začínající klíčovým slovem *using* importuje jmenné prostory z knihoven, které chceme používat v našem programu. Těchto jmenných prostorů můžeme používat více, v takovém případě každý jmenný prostor zapíšeme na samostatný řádek vždy začínající klíčovým slovem *using*. Jmenný prostor můžeme chápat jako pojmenovanou část projektu.

Další část kódu začíná klíčovým slovem *namespace*, kterým definujeme svůj vlastní jmenný prostor. Ve výchozím stavu je název jmenného prostoru roven názvu projektu, ale můžeme ho přepsáním změnit. Pod tímto řádkem následuje znak levé množinové závorky „{“, která označuje začátek bloku kódu, v tomto případě začátek jmenného prostoru MojeAplikace. Každý blok kódu je vždy ukončen znakem pravé množinové závorky „}“, viz poslední znak ve zdrojovém kódu. Pro přehlednost je blok kódu, tj. sobě odpovídající množinové závorky zvýrazněn svislou přerušovanou čarou.



Obrázek 1.5: Blok kódu

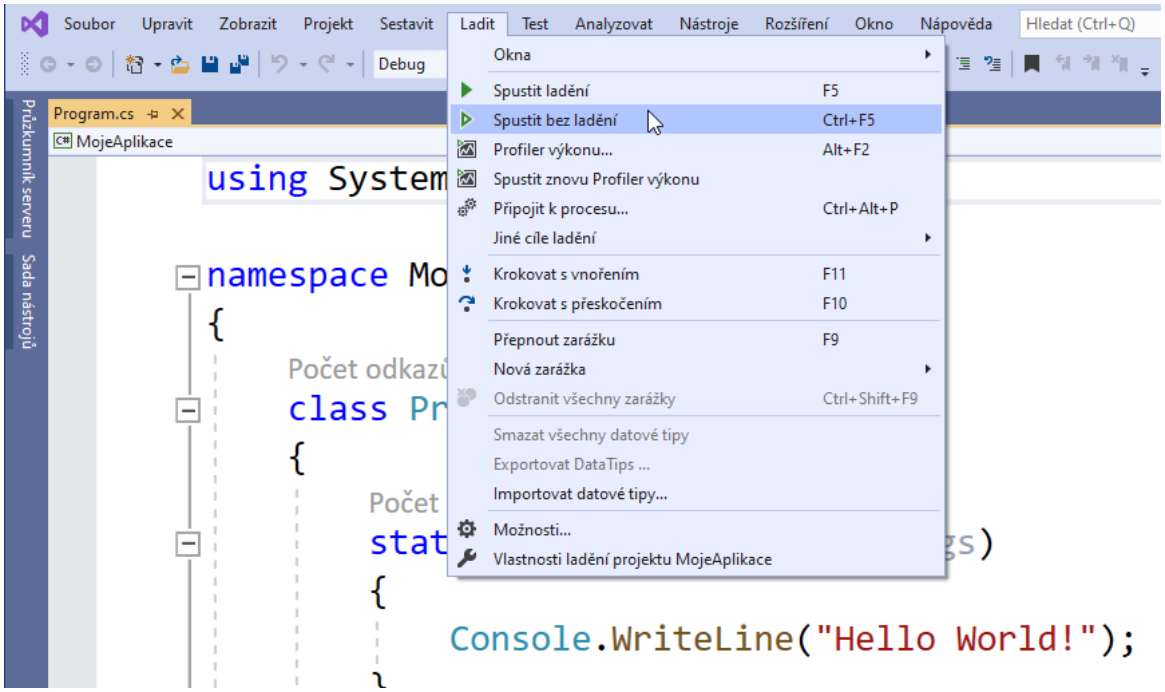
Uvnitř jmenného prostoru je potom definována třída *Program* pomocí klíčového slova *class*. Tento řádek označuje hlavní program našeho projektu. A znovu blok kódu patřící do hlavního programu je uzavřen v množinových závorkách. Poslední část kódu začínající *static void Main* definuje hlavní metodu, která je vstupním bodem programu, tj. při spuštění našeho projektu se začne provádět kód, který je zapsán v metodě *Main*. Jakýkoliv další kód v projektu se může spustit pouze tak, že je zavolán z této metody *Main*.

Všimněme si, že v metodě *Main* je již řádek

Console.WriteLine(“Hello World!”);

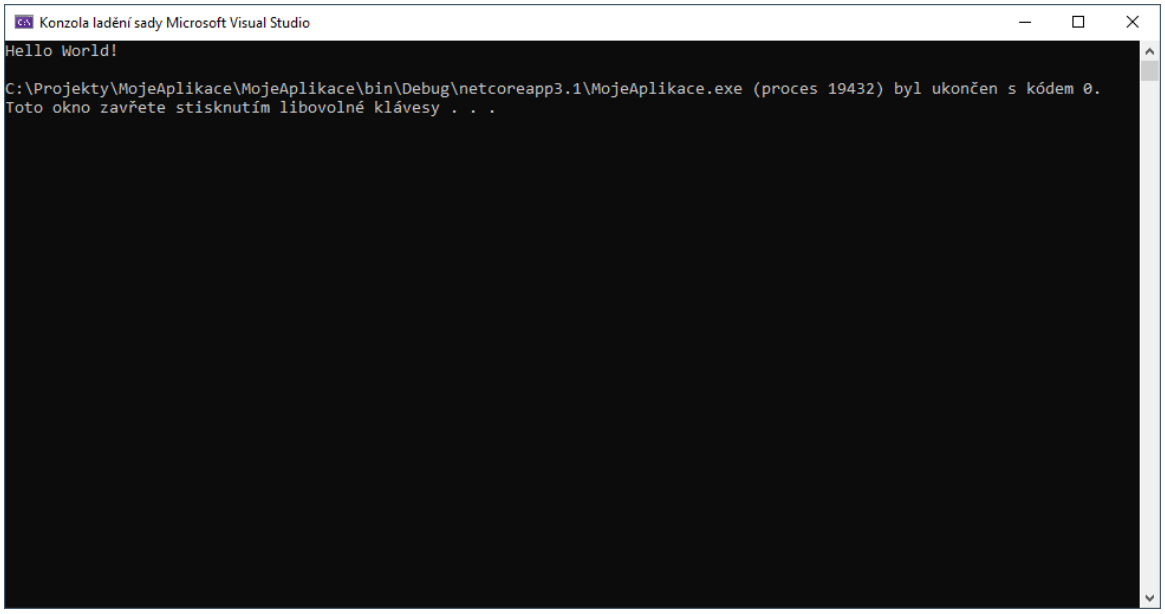
Tento řádek vypíše po spuštění programu na obrazovku text *Hello World!*, kterýžto se vžil pro prezentaci fungování každého programovacího jazyka. Při vytváření vlastní aplikace samozřejmě tento řádek můžeme smazat.

Nyní bychom chtěli náš program spustit. Toto provedeme pomocí menu *Ladit* → *Spustit bez ladění*.



Obrázek 1.6: Spuštění programu

Po potvrzení se nám objeví nové okno s tzv. konzolou, tedy oknem umožňující pouze textový výstup a na něm by měl být zobrazen text *Hello World!*



Obrázek 1.7: Výstup programu

Tímto jsme úspěšně vytvořili a spustili náš první program v jazyce C#. Vzhledem k tomu, že projekty budeme spouštět velmi často je vhodné zapamatovat si klávesovou zkratku *Ctrl + F5*.

1.3.3 KOMENTÁŘE

Pokud vytváříme nějakou jednoduchou aplikaci, je z kódu obvykle patrné, co daná aplikace dělá a jak funguje. Jiná je ovšem situace, pokud je náš program většího rozsahu, zvláště pokud na daném programu pracuje více programátorů. V takovém případě může být orientace v programu náročná, a proto je vhodné používat při psaní kódu komentáře. Komentáře jsou části kódu, které slouží pouze pro naši potřebu a překladačem jsou ignorovány. Existují dva základní druhy komentářů, a to jednořádkové a víceřádkové. Jednořádkové používáme tehdy, pokud je komentář krátký a vleze se nám na jeden řádek. Takový komentář začíná dvěma symboly lomítka //. Vše, co napíšeme za //, je překladačem ignorováno až na konec řádku. Pro přehlednost jsou komentáře v prostředí MS Visual Studia zobrazeny zeleně.

Pokud chceme okomentovat určitou část kódu podrobněji, je výhodné použít komentáře víceřádkové, které začínají dvojicí symbolů lomítka a hvězdičky /* a končí stejnou dvojicí symbolů, ale v opačném pořadí, tj. */. Zde můžeme vidět příklad jednořádkového a víceřádkového komentáře.

```
Počet odkazů: 0
static void Main(string[] args)
{
    Console.WriteLine("Hello World!");  //Výpis textu na obrazovku

    /*
     * Toto
     * je
     * víceřádkový komentář
     */
}
```

Obrázek 1.8: Komentáře

Komentáře však mají i jiné využití než jen popis kódu. Často se využívají i pro dočasné odstranění či zablokování části kódu nebo při úpravách kódu, aby byl vidět původní kód.

1.3.4 LADĚNÍ

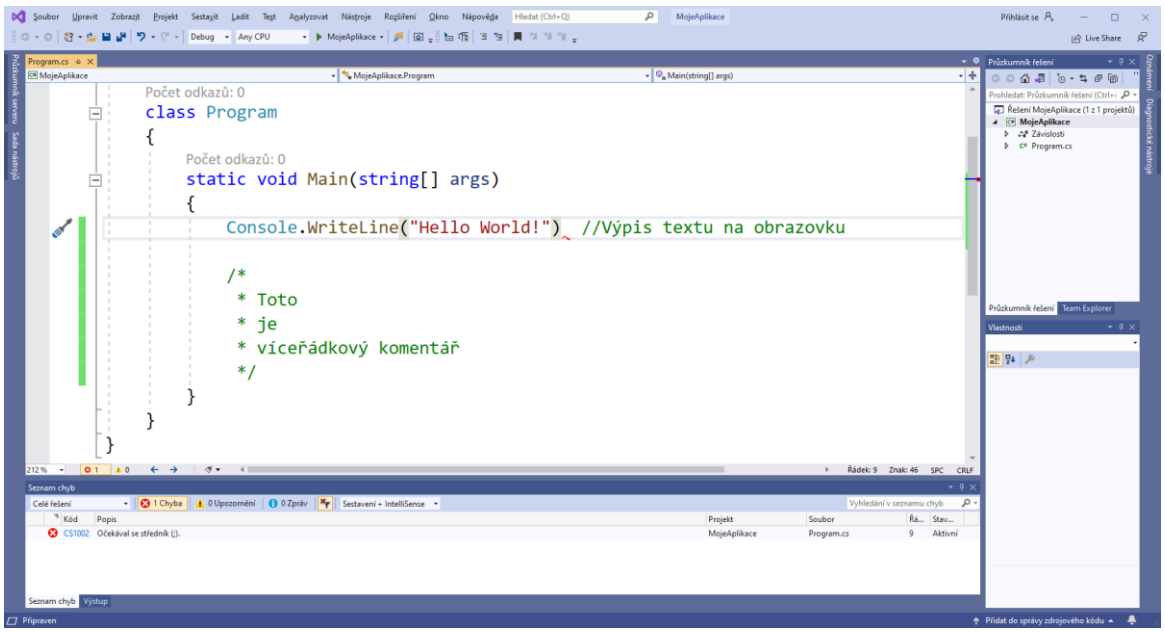
Mohlo by se zdát, že hlavní náplní práce programátora je psaní kódu. Ve skutečnosti tato činnost zabírá pouze menší část času. Mnohem časově náročnějším úkolem je hledání a odstraňování chyb. Rozlišujeme dva základní druhy chyb v programu, a to chyby syntaktické a sémantické.

Začněme chybami syntaktickými, které jsou mnohem jednodušší k nalezení i odstranění, protože MS Visual Studio nám oznámí nejen, že v programu jsou syntaktické chyby, ale dokonce nám oznámí i kde přesně v kódu se nacházejí a často i, jak je máme opravit,

byť tato rada nemusí být vždy správná. Typická syntaktický chyba je např. zapomenutý středník za příkazem. Zkusme si tuto chybu demonstrovat tak, že odstraníme středník za příkazem *Console.WriteLine*. Můžeme si všimnout, že na místě, kde byl středník se nám zobrazí červená vlnovka a při najetí myši na tuto vlnovku se nám zobrazí nápověda s popisem chyby.

Obrázek 1.9: Syntaktická chyba

Pokud bychom si této červené vlnovky nevšimli a pokusili se náš program spustit, zobraz se nám v dolní části obrazovky okno Seznam chyb, kde jsou zobrazeny všechny syntaktické chyby v programu.



Obrázek 1.10: Seznam chyb

Pokud klikneme na popis chyby v seznamu chyb přesune se nám kurzor na místo, kde se daná chyba nachází. Pokud je v programu více chyb, vždy odstraňujeme chyby v pořadí,

v jakém jsou uvedeny v seznamu chyb. Po opravení všech syntaktických chyb můžeme program spustit.

Druhým typem chyb jsou chyby sémantické, které se vyznačují tím, že program jde spustit, ale jeho výstup je jiný, než jsme očekávali. Abychom si mohli demonstrovat tuto chybu, musíme náš úvodní program mírně rozšířit a to tak, že si budeme definovat celočíselnou proměnnou *i* s výchozí hodnotou 5. Poté k této proměnné přičteme číslo 4 a následně tuto proměnnou *i* ještě vynásobíme číslem 3. Nakonec vypíšeme obsah proměnné *i* na obrazovku. Výsledný kód bude vypadat takto.

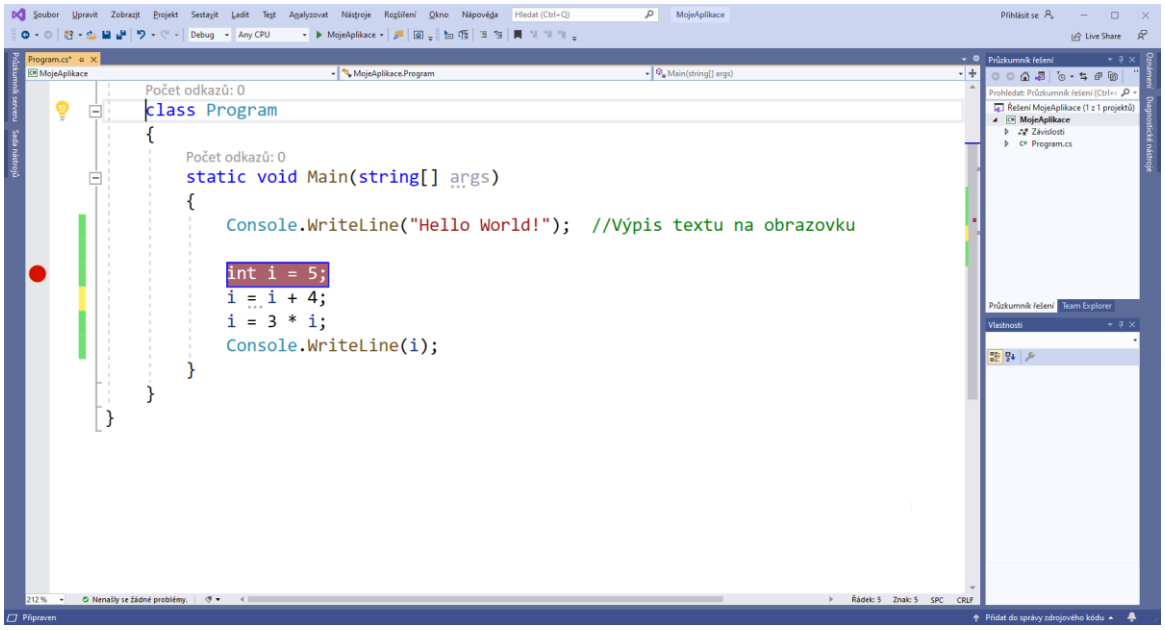
```
Počet odkazů: 0
static void Main(string[] args)
{
    Console.WriteLine("Hello World!");  //Výpis textu na obrazovku

    int i = 5;
    i = i + 4;
    i = 3 * i;
    Console.WriteLine(i);
}
```

Obrázek 1.11: Syntaktická chyba

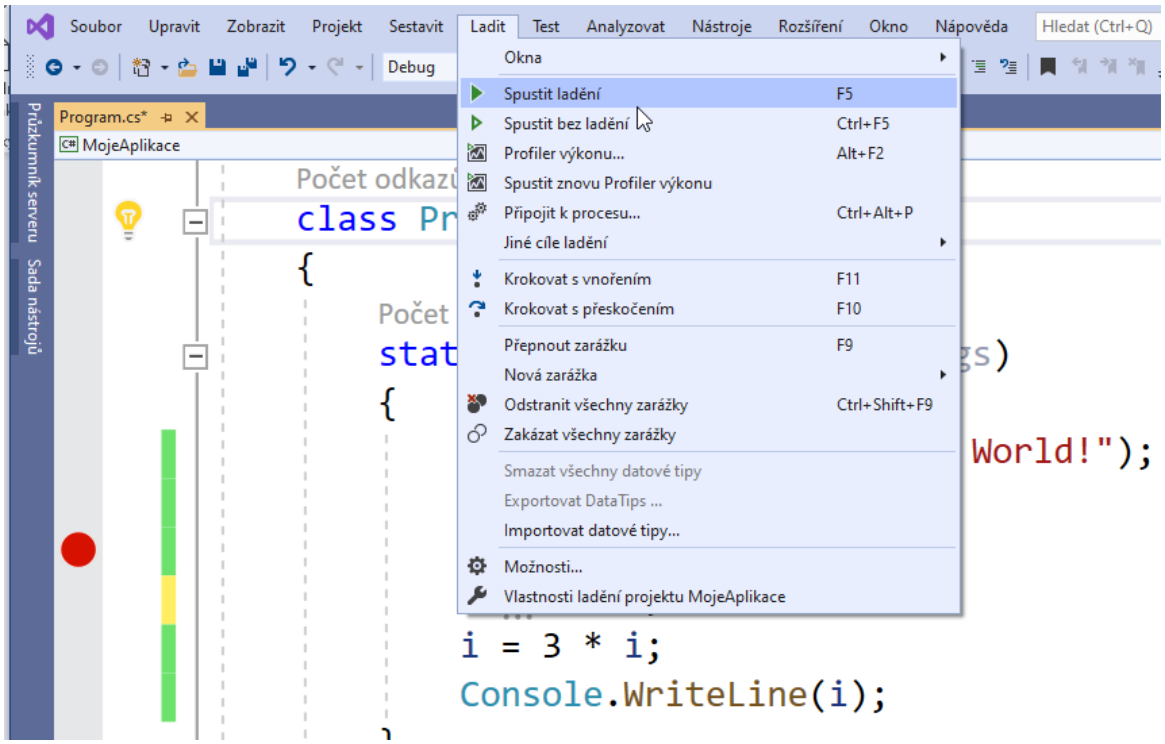
Po spuštění programu se nám na výstupní konzoli zobrazí hodnota 27, ovšem my jsme očekávali, že by výstupem mělo být číslo 18. Z výš uvedeného jednoduchého kódu je zřejmé, jakým způsobem jsme dostali výsledek 27 a jak můžeme kód opravit, aby výsledkem bylo číslo 18. Ovšem v praxi bude kód mnohem delší a komplikovanější, a tudíž na první pohled nebude zřejmě, proč nám program poskytuje jiný výsledek, než jsme očekávali.

Obecný postup pro hledání sémantických chyb se nazývá *ladění* nebo *debugging*. Základem je vložit do programu bod *přerušení* nebo též *breakpoint*. Nejprve ale musíme přibližně odhadnout, kde se může příslušná chyba nacházet. V našem případě to bude někde v místě, kde se pracuje s proměnnou *i*, ale nevíme, kde přesně, takže vložíme bod přerušení na místo prvního výskytu proměnné *i*. Samotný bod přerušení vložíme do kódu kliknutím do levého šedého pruhu u příslušného řádku, čímž se nám příslušný řádek zabarví červeně a v levém šedém pruhu se zobrazí červené kolečko.



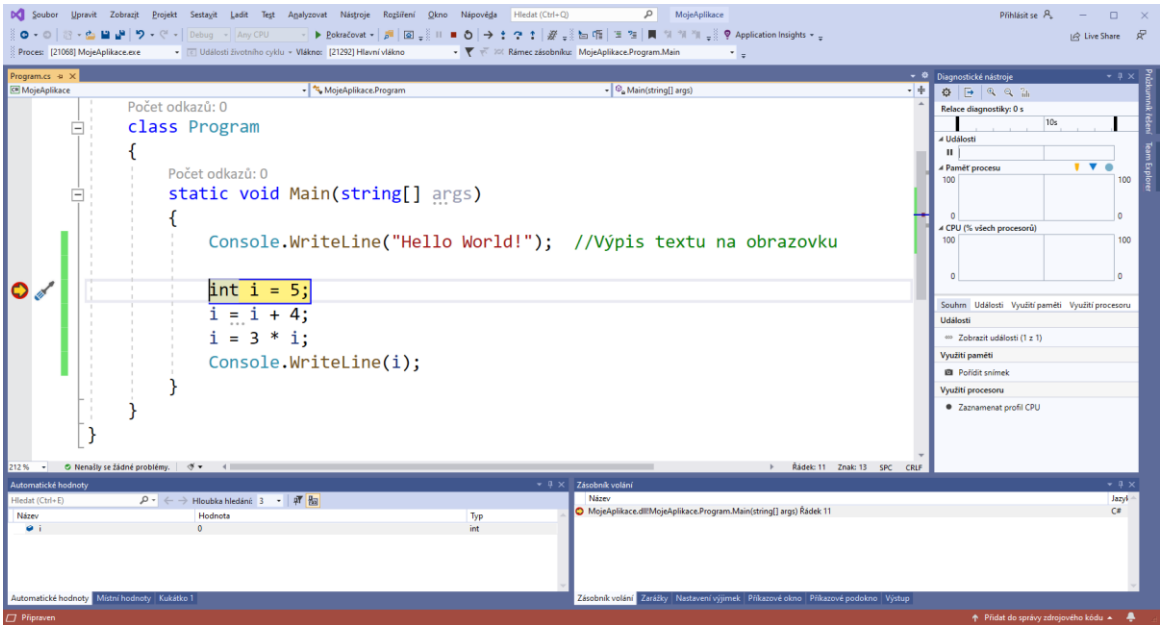
Obrázek 1.12: Bod přerušení

Pokud nyní spustíme program stejným způsobem, jako jsme to dělali doposud, celý program proběhne a ihned skončí. Abychom spustili ladění programu, musíme v menu zvolit *Ladit → Spustit ladění*.



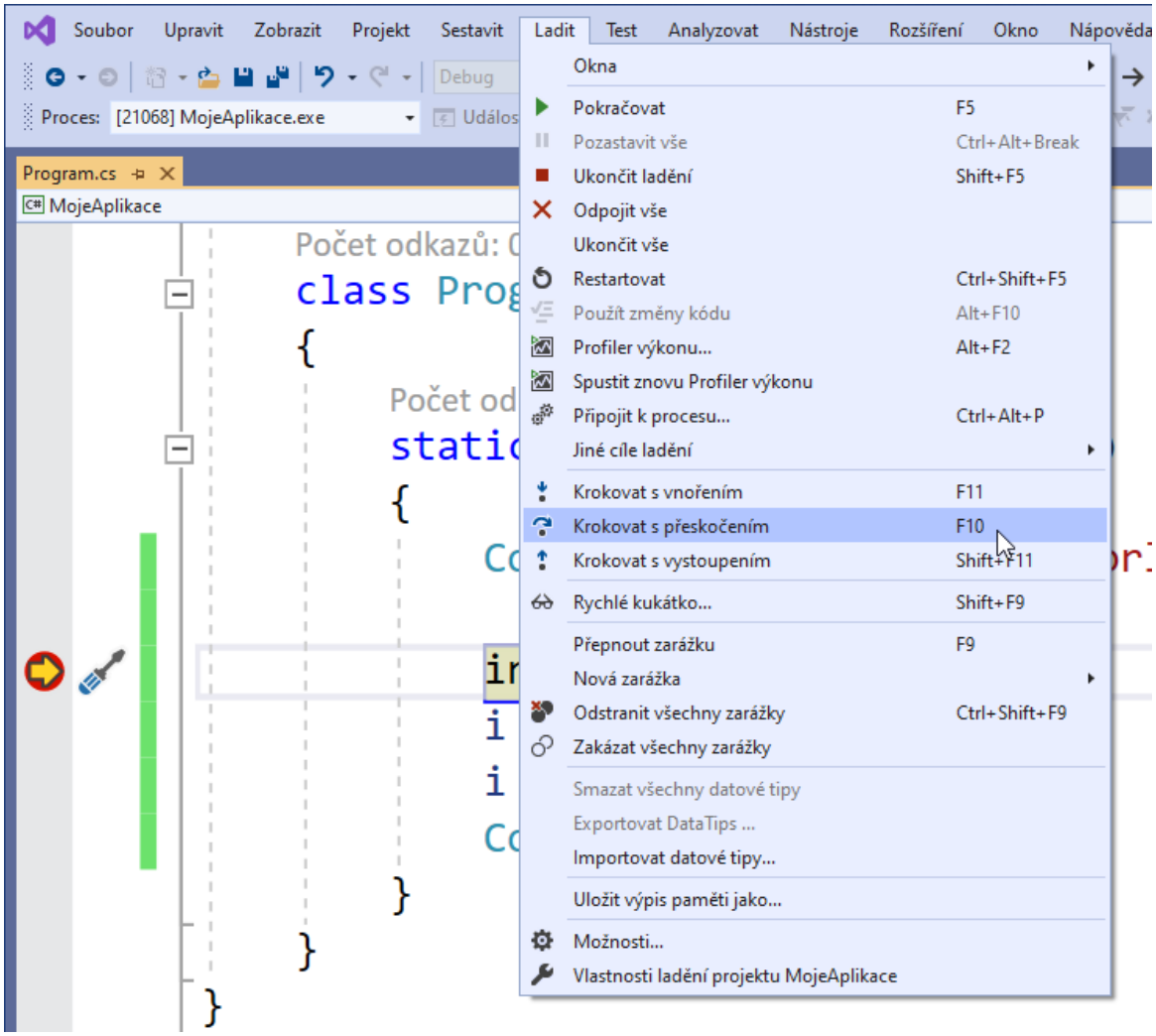
Obrázek 1.13: Ladění

Nyní se nám běh programu přeruší v nastaveném bodě přerušení, což poznáme tak, že se řádek, kde se přerušil běh programu, zvýrazní žlutě.



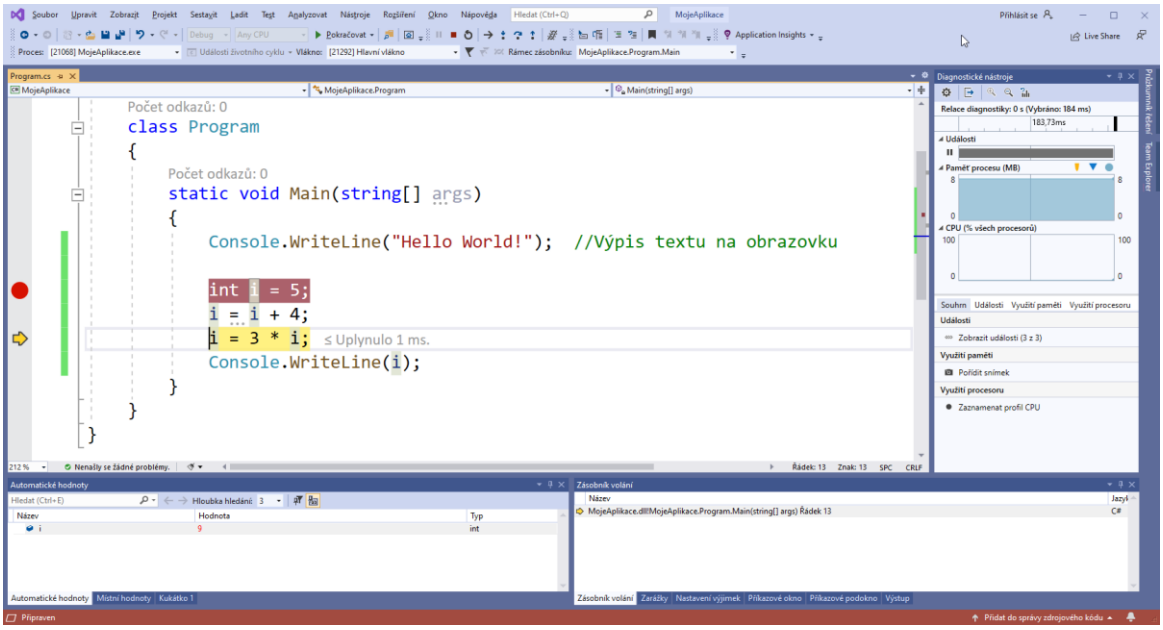
Obrázek 1.14: Ladění

Zároveň si můžeme všimnout, že se nám změnilo rozložení oken tak, aby se zobrazily informace důležité pro ladění. V pravém horním okně *Diagnostické nástroje* můžeme vidět, kolik náš program zabírá paměti, či kolik spotřebovává procent procesoru. V pravém dolním okně *Zásobník volání* můžeme vidět, z jaké metody byl zavolán aktuální příkaz. Nás však bude v tuto chvíli nejvíce zajímat levé dolní okno *Automatické hodnoty*, které nám zobrazuje hodnoty proměnných v aktuálním kontextu. V našem programu je definována pouze jedna proměnná i, která je rovněž zobrazena v tomto okně. Ve sloupci Hodnota můžeme vidět, že její hodnota je 0, je tomu tak proto, že příkaz přiřazení $i = 5$ ještě nebyl vykonán. Pomocí menu *Ladit* → *krokovat s přeskočením* můžeme vykonat jeden příkaz za bodem přerušení.



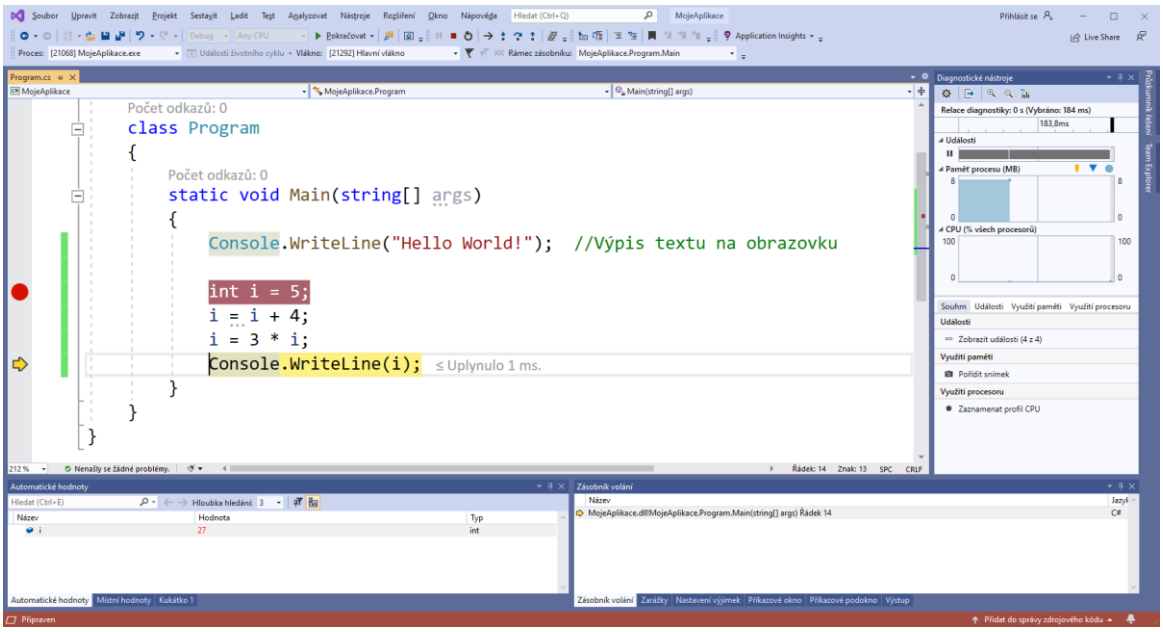
Obrázek 1.15: Krokování 1

Vykonáním příkazu *int i = 5* došlo přiřazení hodnoty 5 do proměnné *i*, což můžeme vidět v okně Automatické hodnoty. Všimněte si rovněž, že hodnota 5 je zobrazena červeně, čímž je zvýrazněno, že v daném kroku došlo ke změně hodnoty této proměnné. Toto zejména oceníme, pokud je v daném kontextu definováno více proměnných. Zároveň si můžeme všimnout, že je nyní zvýrazněn následující řádek, což nám ukazuje, který řádek bude vykonán v následujícím kroku. Nyní provedeme znovu krokování s přeskočením buď pomocí menu nebo klávesou *F10*.



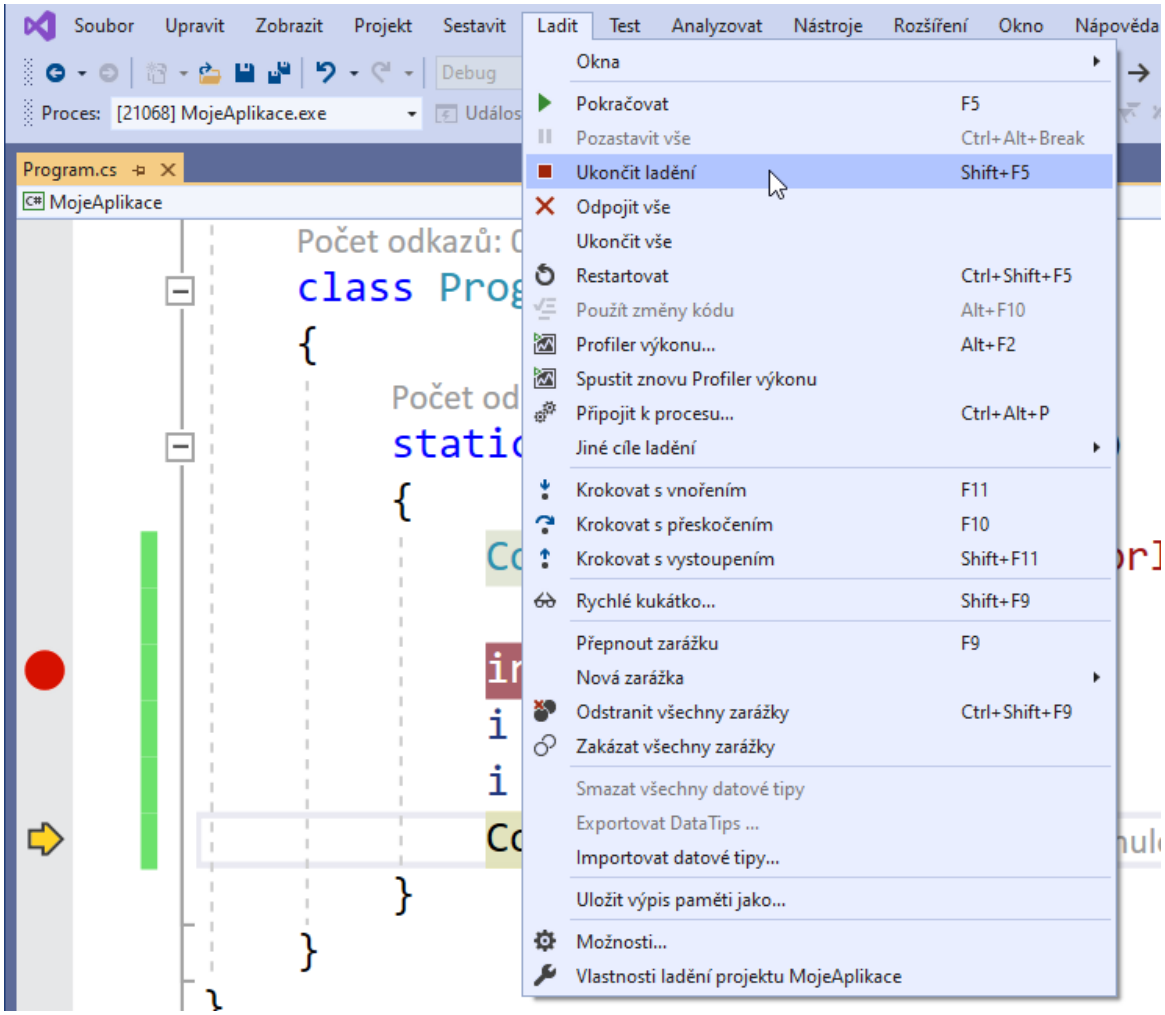
Obrázek 1.16: Krokování 2

Tímto jsme vykonali příkaz přiřazení i = i + 4. V okně Automatické hodnoty vidíme, že hodnota proměnné i je nyní 9. Teď ještě jednou provedeme krokování s přeskočením.



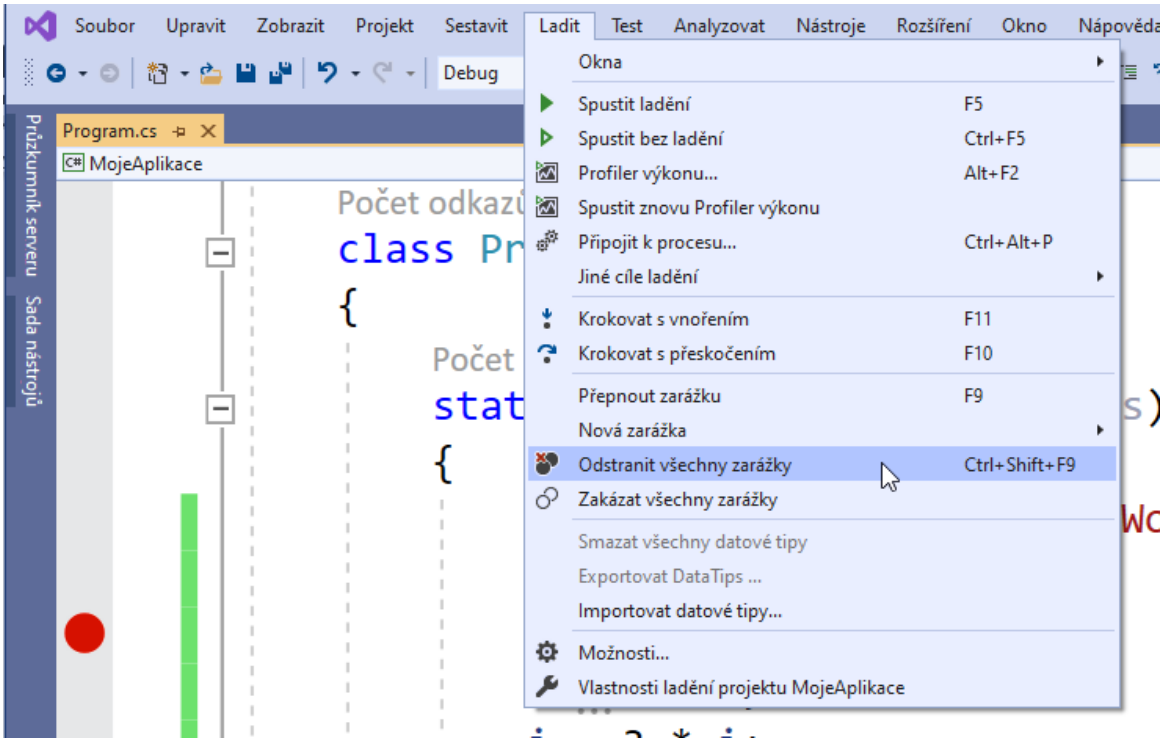
Obrázek 1.17: Krokování 3

Provedením dalšího příkazu vidíme, že nyní je hodnota proměnné *i* 27. Jelikož by správně měla být nyní hodnota proměnné *i* 18, zjistili jsme, že v posledním řádku, který jsme vykonali je chyba a místo násobení číslem 3 by tam mělo být násobení číslem 2. Můžeme tedy ukončit ladění pomocí menu Ladit → Ukončit ladění.



Obrázek 1.18: Ukončení ladění

Po ukončení ladění můžeme program editovat a opravit chybu. Na závěr je ještě vhodné odstranit všechny body přerušení, což provedeme opětovným kliknutím na červené kolečko v levém šedém pruhu nebo v menu *Ladit → Odstranit všechny zarážky*.



Obrázek 1.19: Odstranění bodu přerušení

OTÁZKY

1. Jazyk C# vychází z jazyka
   a. Basic
   b. C
   c. Prolog
2. Jazyk C# je
   a. Kompilovaný jazyk
   b. Interpretovaný jazyk
   c. Jazyk s virtuálním strojem
3. Pro komentáře v jazyce C# používáme symbol
   a. //
   b. \\
   c. –
4. Pokud je v programu syntaktická chyba potom
   a. Program lze spustit, ale výstup programu je jiný než očekávaný.
   b. Při pokusu o spuštění programu se MS Visual se zhroutí.
   c. Program nelze spustit a MS Visual Studio nám zobrazí seznam syntaktických chyb v programu.
5. Pokud je v programu sémantická chyba potom
   a. Program lze spustit, ale výstup programu je jiný než očekávaný.

   b. Při pokusu o spuštění programu se MS Visual se zhroutí.
   c. Program nelze spustit a MS Visual Studio nám zobrazí seznam syntaktických chyb v programu.
6. Sémantické chyby obvykle hledáme pomocí
   a. Křišťálové koule
   b. Ladění
   c. Doplnění středníku
7. Místo v programu, kde se zastaví provádění programu při ladění se nazývá
   a. Podmínka
   b. Operand
   c. Bod přerušení
8. Proces, při kterém postupně vykonáváme jednotlivé příkazy se nazývá
   a. Krokování
   b. Spuštění
   c. Inspekce
9. Který příkaz vypíše na obrazovku text?
   a. Console.Writeline
   b. Console.writeline
   c. Console.WriteLine
10. Jak se v jazyce C# ukončuje příkaz?
   a. Novým řádkem
   b. Středníkem
   c. Tečkou

SHRNUTÍ KAPITOLY

Tato kapitola obsahovala základy programování. Seznámili jsme se se stručnou historií programování, dále jsme si ukázali, jak nainstalovat Microsoft Visual Studio a konečně jsme vytvořili svůj první program v jazyce C#. Pro lepší čitelnost jsme si ukázali, jak náš kód okomentovat. Nakonec jsme se podívali na rozdíl mezi syntaktickými a sémantickými chybami a zejména jsme si ukázali, jak chyby v programu najít a odstranit.

ODPOVĚDI

1. b
2. c
3. a
4. c
5. a

6. b
7. c
8. a
9. c
10. b

2 TYPY DAT A JEJICH REPREZENTACE

RYCHLÝ NÁHLED KAPITOLY

Tato kapitola je zaměřena na způsob, jakým můžeme ukládat dočasné informace do paměti počítače. Seznámíme se s termínem proměnná a uvedeme si jejich druhy, tedy datové typy. Zároveň se ukážeme použití základních operací s čísly v jazyce C#.

CÍLE KAPITOLY

Po prostudování této kapitoly budete umět:

- Definovat rozdíl mezi proměnnou a datovým typem.
- Jak deklarovat proměnnou a jak s ní v programu dále pracovat.
- Vyjmenovat základní datové typy.
- Definovat vlastní uživatelské datové typy.
- Převádět proměnné různých datových typů mezi sebou.
- Provádět základní operace s čísly.

KLÍČOVÁ SLOVA KAPITOLY

Proměnná, konstanta, paměť, datový typ, konverze, přetypování, operace.

2.1 Proměnné

Proměnná je v programování označení pro identifikátor (symbolické jméno), které uchovává určitou informaci při běhu programu. Proměnná může nabývat známých nebo neznámých informací, které se nazývají hodnota. Jméno proměnné je obvyklá cesta k získání reference k uložené hodnotě v paměti počítače. Separace jména a obsahu umožňuje použít jméno, které se používá nezávisle na přenesené informaci, kterou zastupuje. Identifikátor ve zdrojovém kódu zastupuje nějakou hodnotu, která se za běhu programu může měnit.

Proměnné v programování nemusejí přímo korespondovat s konceptem proměnných v matematice. Hodnota v počítačové proměnné nemusí být nezbytně část matematické rovnice či vzorce. Proměnné můžou být použity v opakujících se procesech: přiřazena hodnota

v jednom místě programu, poté použita v jiné části programu a pak může být přiřazena úplně nová hodnota a celý proces se může opakovat stejnou cestou (iterace). Proměnným v počítačovém programování jsou většinou nazývány dlouhými názvy tak, aby bylo z názvu patrné k čemu slouží, na rozdíl od proměnných v matematice, kde jsou obvykle nazvány jedním či dvěma znaky pro jasnou stručnost zápisu.

Na rozdíl od matematických výrazů, v programování můžou proměnné mít zpravidla více znaků, jako například „COUNT“ nebo "total". Proměnné, jejichž název má pouze jeden znak, jsou často používány jako iterační (například i, j, k) nebo proměnné používané jako index pole.

Některé jmenné konvence jsou již součástí syntaxe jazyka a ovlivňují platnost proměnné. Skoro ve všech jazycích nemohou jména proměnných začínat číslem či nemohou obsahovat mezery. Některé jazyky současně neumožňují, aby proměnná začínala interpunkčním znakem (jediným platným znakem bývá podtržítko („_“). Interpunkční znaky se nemohou objevovat ani nikde v názvu proměnné.

Rozlišování malých a velkých písmen se také liší jazyk od jazyku. Některé (zpravidla starší) programovací jazyky nerozlišují použitou velikost písmen, avšak většina moderních jazyků včetně C# ano (pro příklad proměnná „pocet“ a „Pocet“ se liší).

Ve výsledku však bývá rozlišování malých a velkých písmen pouze otázkou vkusu či programátorského stylu. Na úrovní strojového kódu nejsou proměnné používány vůbec, takže používání proměnných je pouze pomůckou pro programátory, aby učinily kód čistším a snadnějším na pochopení. Použitím nevhodně nazvaných proměnných se kód stává hůře čitelným.

Název proměnné v jazyce C# se může skládat z písmen, číslic a podtržítek, ale nesmí začínat číslicí. Pokud je název proměnné víceslovný, je vhodné každé další slovo začít velkým písmenem. Doporučuje se v názvech proměnných používat pouze písmena anglické abecedy, a vyhnout se tedy např. používání češtiny s diakritikou. Zároveň platí, že proměnná nesmí mít stejný název jako klíčová slova jazyka C# a nesmí se jmenovat jako již existující datové typy. Nelze tedy proměnnou pojmenovat *Console, int, float* apod.

K ZAPAMATOVÁNÍ

Je vhodné proměnné nazývat podle toho, jaká data obsahují. Čím budou vaše programy rozsáhlejší, tím více oceníte vhodné názvy proměnných, které vám usnadní orientaci ve zdrojovém kódu.

S pojmem proměnná souvisí i pojem konstanta, kdy chceme, aby proměnné šlo přiřadit hodnotu pouze v místě její deklarace, ale později nikde v kódu nechceme, aby bylo možné

tuto hodnotu změnit. A pokud se takovouto konstantu přesto pokusíme změnit, překladač nám oznámí syntaktickou chybu. Konstantu definujeme tak, že před deklaraci proměnné zapíšeme klíčové slovo *const*.

```
const double cisloPI = 3.14;
cisloPI = 5;  //Pokus o změmu konstanty - syntaktická chyba
```

Obrázek 2.1: Konstanta

2.2 Datové typy

Datový typ definuje v programování druh nebo význam hodnot, kterých smí nabývat proměnná (nebo konstanta). Datový typ je určen oborem hodnot a zároveň výpočetními operacemi, které lze s hodnotami tohoto typu provádět.

2.2.1 ZÁKLADNÍ DATOVÉ TYPY

Součástí programovacího jazyka je definice základních datových typů. Pomocí těchto základních typů může ve většině jazyků programátor tvořit nové složené typy. V jazyce C# jsou definovány tyto základní datové typy.

- *byte* – celočíselná hodnota bez znaménka, zabírá v paměti 1 bajt, má rozsah 0 až 255
- *sbyte* – celočíselná hodnota se znaménkem, zabírá v paměti 1 bajt má rozsah -128 až 127
- *short* – celočíselná hodnota se znaménkem, zabírá v paměti 2 bajty, má rozsah -32768 až 32767
- *ushort* – celočíselná hodnota bez znaménka, zabírá v paměti 2 bajty má rozsah 0 až 65535
- *int* – celočíselná hodnota se znaménkem, zabírá v paměti 4 bajty, má rozsah – 2 147 483 648 až 2 147 483 647
- *uint* – celočíselná hodnota bez znaménka, zabírá v paměti 4 bajty má rozsah 0 až 4 294 967 295
- *long* – celočíselná hodnota se znaménkem, zabírá v paměti 8 bajtů, má rozsah -9 223 372 036 854 775 808 až 9 223 372 036 854 775 807
- *ulong* – celočíselná hodnota bez znaménka, zabírá v paměti 8 bajtů má rozsah 0 až 18 446 744 073 709 551 615
- *float* – reálná hodnota se znaménkem, zabírá v paměti 4 bajty, má rozsah +/-1.5 * 10^{-45} až +/-3.4 * 10^{38}
- *double* – reálná hodnota se znaménkem, zabírá v paměti 8 bajtů, má rozsah +/-5.0 * 10^{-324} až +/-1.8 * 10^{308}
- *decimal* – reálná hodnota se znaménkem, zabírá v paměti 16 bajtů, používá se pro přesné výpočty s 28-29 platnými číslicemi

- *char* – znak v kódování Unicode, zabírá v paměti 2 bajty
- *bool* – logická hodnota true/false, zabírá v paměti 1 bajt
- *string* – textový řetězec, zabírá v paměti proměnnou délku podle počtu znaků v řetězci

Zde můžete vidět příklady deklarace proměnných všech výše uvedených datových typů s nastavením výchozí hodnoty.

```
byte a = 213;
sbyte b = -124;
short c = -32;
ushort d = 56;
int e = -5;
uint f = 6;
long g = -42;
ulong h = 51;
float i = 2.3f;
double j = -4.1;
decimal k = 155.4m;
char l = 'A';
bool m = true;
string n = "Ahoj";
```

Obrázek 2.2: Datové typy

Všimněte si, že u přiřazení hodnoty do proměnné typu *float* píšeme za číslem znak *f*. Je tomu tak proto, že defaultně jsou přímo zapsaná čísla v kódu považována za typ double a znakem *f* dojde k přetypování na *float*, pokud bychom na tuto skutečnost zapomněli, překladač by nám oznámil syntaktickou chybu. Ze stejného důvodu píšeme u přiřazení hodnoty do proměnné *decimal* za číslem znak *m*. Dále si všimněte, že u přiřazení znaku do proměnné typu *char* zadáváme tento znak do apostrofů na rozdíl od proměnné typu *string*, kde řetězec zadáváme v uvozovkách.

U číselných datových typů někdy potřebujeme přiřazovat proměnnou jednoho datového typu do proměnné jiného datového typu. Pokud se například pokusíme přiřadit proměnnou typu *int* do proměnné typu double, příkaz proběhne úspěšně. Pokud ovšem se o totéž pokusíme v opačném pořadí, tj. přiřadit proměnnou typu double do proměnné typu *int*, potom nám překladač oznámí syntaktickou chybu. Jedná se o určitou ochranu, aby tímto přiřazením nedošlo ke ztrátě informace, tj. desetinné části čísla. Pokud ale přesto chceme takové přiřazení explicitně provést, musíme proměnnou typu *double* přetypovat na *int* a to tak, že

výsledný datový typ *int* napíšeme do kulatých závorek před příslušnou proměnnou typu *double*. Zde se můžete podívat na několik možností přiřazení různých datových typů.

```
double cisloDouble = 134.7;
int cisloInt = (int)cisloDouble;  //cisloInt bude 134
float cisloFloat = (float)cisloDouble;
decimal cisloDecimal = (decimal)cisloDouble;
byte cisloByte = (byte)cisloDouble;  //cisloInt bude 134
short cisloShort = (short)cisloDouble;  //cisloInt bude 134
long cisloLong = (long)cisloDouble;  //cisloInt bude 134
```

Obrázek 2.3: Přetypování číselných datových typů

Všimněte si, že přiřazením reálného čísla do proměnné celočíselného datového typu nedojde k zaokrouhlení reálného čísla na celé číslo, ale dojde k odříznutí desetinné části čísla. Na to je třeba pamatovat.

V řadě případů budeme ovšem potřebovat převádět mezi sebou nejen číselné datové typy, ale i typ *string* na nějaký číselný datový typ, zejména tehdy pokud budeme načítat nějaký vstup od uživatele. Pro tento typ převodu používáme metody začínající na *Convert.To*, např. *Convert.ToInt32* nebo *Convert.ToDouble*. Níže se můžete podívat na různé převody textových hodnot na číslo.

```
string cisloString = "3";
double cisloPrevedeneDouble = Convert.ToDouble(cisloString);
int cisloPrevedeneInt = Convert.ToInt32(cisloString);
float cisloPrevedeneFloat = Convert.ToSingle(cisloString);
decimal cisloPrevedeneDecimal = Convert.ToDecimal(cisloString);
byte cisloPrevedeneByte = Convert.ToByte(cisloString);
short cisloPrevedeneShort = Convert.ToInt16(cisloString);
long cisloPrevedeneLong = Convert.ToInt64(cisloString);
```

Obrázek 2.4: Převody řetězce na čísla

Všimněte si, že názvy některých konverzních metod ne vždy odpovídají výslednému typu, např. pro *float* je použita metoda *Convert.ToSingle*, nebo pro celočíselné datové typy je v názvu konverzní metody počet bitů, který příslušný typ zabírá v paměti.

Kromě konverze z řetězce na číslo je mnohdy zapotřebí provést konverzi opačným směrem. Pokud chceme číslo převést na řetězec, použijeme k tomu metodu *ToString*. Tuto metodu budeme používat následujícím způsobem. Napíšeme název proměnné, jejíž hodnotu chceme převést, a na ni zavoláme metodu *ToString*. Nyní si ukážeme příklady převodů čísel na řetězec, budeme k tomu používat číselné proměnné z předchozího příkladu.

```
string retezecDouble = cisloPrevedeneDouble.ToString();
string retezecInt = cisloPrevedeneInt.ToString();
string retezecFloat = cisloPrevedeneFloat.ToString();
string retezecDecimal = cisloPrevedeneDecimal.ToString();
string retezecByte = cisloPrevedeneByte.ToString();
string retezecShort = cisloPrevedeneShort.ToString();
string retezecLong = cisloPrevedeneLong.ToString();
```

Obrázek 2.5: Převody čísel na řetězce

2.2.2 VÝČTOVÝ DATOVÝ TYP

Výčtový datový typ tvořený konečnou omezenou množinou pojmenovaných hodnot. Každý člen výčtového typu je tvořen identifikátorem a hodnotou. Hodnotou zpravidla bývá celočíselná konstanta. Výčtový typ je zvláštní hodnotový typ s předem určenou sadou číselných hodnot. V základním nastavení jsou tyto číselné hodnoty celá čísla (typ *int*), ale mohou být i typu *long*, *byte* atd. Identifikátor se obvykle v kódu chová jako konstanta.

Výčtový typ v C# je uvozen klíčovým slovem *enum*. Před klíčovým slovem *enum* může být klasifikátor přístupu. Při přiřazování ordinálních hodnot se defaultně začíná od nuly. Ukážeme si to na příkladu dnů v týdnu.

```
enum Dny
{
    Pondeli,
    Utery,
    Streda,
    Ctvrtek,
    Patek,
    Sobota,
    Nedele
}
```

Obrázek 2.6: Výčtový datový typ

V tomto případě *Dny.Pondeli* bude reprezentována hodnotou 0, *Dny.Utery* hodnotou 1, *Dny.Streda* hodnotou 2, atd. Avšak toto není úplně ideální, protože při číslování dnů v týdnu máme zažito, že pondělí je první nikoli nultý den v týdnu, je proto potřeba počítat od 1. Toho je možné docílit přidělením jedničky prvnímu identifikátoru. V C# se hodnoty identifikátorům udělují s operátorem =. Upravený výčtový datový typ *Dny* bude vypadat následovně.

```
enum Dny
{
    Pondeli = 1,
    Utery,
    Streda,
    Ctvrtek,
    Patek,
    Sobota,
    Nedele
}
```

Obrázek 2.7: Výčtový datový typ 2

K ZAPAMATOVÁNÍ

Používání výčtového datového typu zlepšuje čitelnost kódu.

2.2.3 UŽIVATELSKY DEFINOVANÝ DATOVÝ TYP

V některých případech nám základní datové typy nebudou stačit. Například pokud budeme chtít uchovat více údajů o dané osobě, jako jméno příjmení a věk. Samozřejmě bychom mohli definovat tři samostatné proměnné a každý údaj zapsat do samostatné proměnné. Problém by ale mohl nastat, kdybych chtěli uchovávat údaje nikoli pouze o jedné osobě. Potom by se taková reprezentace dat stala značně nepřehlednou. Lepší možností je tedy definovat si uživatelsky strukturovaný datový typ, který definujeme pomocí klíčového slova *struct*. Pro náš příklad si můžeme definovat typ *Osoba* následovně.

```
struct Osoba
{
    public string Jmeno;
    public string Prijmeni;
    public int Vek;
}
```

Obrázek 2.8: Deklarace uživatelsky definovaného typu

V programu potom můžeme deklarovat proměnnou nového datového typu *Osoba* a k jednotlivý členům uvnitř typu *Osoba* přistupuje pomocí symbolu tečky.

```
Osoba o;
o.Jmeno = "Pavel";
o.Prijmeni = "Beran";
o.Vek = 35;
Console.WriteLine(o.Jmeno + " " + o.Prijmeni + " " + o.Vek);
```

Obrázek 2.9: Deklarace proměnné typu Osoba

2.3 Základní operace s čísly

V programovacím jazyku C# můžete používat standardní matematické operace, jako je např. sčítání, odčítání, násobení a dělení. Podívejte se, jaké symboly se pro jednotlivé elementární matematické operace používají.

Tabulka 2.1

Matematická operace	Symbol
Sčítání	+
Odčítání	-
Násobení	*
Dělení	/
Zbytek po celočíselném dělení	%

Nyní si ukažme, jak využijeme výše uvedené operace při práci s proměnnými. Na začátku ukázky deklarujeme dvě celočíselné a jednu desetinnou proměnnou. Poté si do dalších proměnných ukládáme výsledky jednotlivých matematických operací. Dále si všimněte, že platí přednost násobení a dělení před sčítáním a odčítáním. Samozřejmě je možné příslušnou operaci uzavřít do závorek a tím ji upřednostnit.

```
int x = 10;
int y = 4;
float z = 4;

int soucet = x + y;
int rozdil = x - y;
int soucin = x * y;

//Pokud dělíme dvě celá čísla, je provedeno celočíselné dělení.
int podilCeleCislo = x / y;  //Výsledek je 2
int zbytekPoCelociselnemDeleni = x % y;  //Zbytek po celočíselném dělení je 2

//Pokud je při dělení alespoň jedno číslo desetinné, je i výsledek desetinné číslo
float podilDesetinneCislo = x / z;  //Výsledek je 2.5

int podilSpatnyTyp = x / z;   //Toto je špatně - syntaktická chyba
int vysledek1 = 2 * 3 + 5 * 4;  //Výsledek je 26
int vysledek2 = 2 * (3 + 5) * 4;  //Výsledek je 64
int vysledek3 = 2 + 18 / 2;  //Výsledek je 11
int vysledek4 = (2 + 18) / 2;  //Výsledek je 10
```

Obrázek 2.10: Základní matematické operace

K ZAPAMATOVÁNÍ

Věnujte pozornost zejména odlišnosti v situaci, kdy dělíme dvě celá čísla a kdy je alespoň jedno z čísel desetinné. Pokud dělíme dvě celá čísla je provedeno celočíselné dělení. Pokud je při dělení alespoň jedno číslo desetinné, je i výsledek desetinný.

OTÁZKY

1. Proměnná je
    a. Pojmenované místo v paměti
    b. Používá se místo komentáře
    c. Druh cyklu
2. Konstanta se deklaruje pomocí klíčového slova
    a. constant
    b. const
    c. unchangeable
3. Při ukládání řetězce do proměnné typu string, musíme řetězec uzavřít do
    a. Apostrofů
    b. Hvězdiček
    c. Uvozovek
4. Proměnná typu bool nemůže nabývat hodnoty
    a. true

   b. false
   c. null
5. Pro operaci zbytek po dělení používáme symbol
   a. %
   b. $
   c. &
6. Jakou hodnotu bude mít proměnná *a*? *double a = 2 / 3*;
   a. 1
   b. 0
   c. 0.6666666
7. Výčtový datový typ definujeme pomocí klíčového slova
   a. enum
   b. list
   c. values
8. Když chceme vynásobit dvě čísla, která jsou uložená v proměnných typu *string*
   a. Nemusíme proměnné převádět, program sám pozná, že proměnné jsou čísla.
   b. Jednu proměnnou musíme převést na číselný datový typ.
   c. Obě proměnné musíme převést na číselný datový typ.
9. Který datový typ nám umožní uložit největší číslo?
   a. int
   b. byte
   c. short
10. Pokud chceme vydělit dvě celočíselné proměnné v oboru reálných čísel
   a. Nemusíme dělat nic, dělení probíhá vždy v oboru reálných čísel.
   b. Alespoň jednu proměnnou musíme přetypovat na reálný datový typ.
   c. Celočíselné proměnné nelze dělit.

SHRNUTÍ KAPITOLY

Tato kapitola byla zaměřena na způsoby uchovávání informací do paměti počítače. Seznámili jsme se s proměnnými, ukázali jsme si, jak můžeme proměnnou deklarovat, jak do ní přiřadit hodnotu a jak hodnoty z proměnných zpět přečíst. Dále jsme si ukázali nejen základní datové typy, ale i některé pokročilejší, jako například výčtový datový typ nebo uživatelsky definovaný datový typ. Rovněž jsme zmínili možnosti konverze jednotlivých datových typů mezi sebou. Nakonec jsme si ukázali základní operace s čísly.

ODPOVĚDI

1. a
2. b
3. c
4. c
5. a
6. b
7. a
8. c
9. a
10. b

3 METODY

RYCHLÝ NÁHLED KAPITOLY

V této kapitole si představíme základní prvek strukturovaného programování, a to členění programu do menších částí, které nám potom umožňují opakované vykonávání částí kódu. Začneme těmi nejjednoduššími metodami, které pokaždé vykonají tutéž činnost, dále si popíšeme způsob, jak mohou metody vracet nějaký výsledek, a nakonec budeme vytvářet metody, které budou měnit své chování v závislosti na hodnotách parametrů, které jim předáme. Ukážeme si rovněž, jaký je rozdíl mezi parametry předávané hodnotou a parametry předávané odkazem.

CÍLE KAPITOLY

Po prostudování této kapitoly budete umět:

- Vytvořit metodu.
- Využívat návratovou hodnotu.
- Modifikovat chování metody pomocí parametrů.
- Rozlišovat mezi parametry volanými hodnotou a odkazem.
- Vracet více než jednu návratovou hodnotu.

KLÍČOVÁ SLOVA KAPITOLY

Metoda, parametr, hodnota, odkaz, návratová hodnota.

Programy, které jsme zatím vytvářeli, byly poměrně krátké, protože sloužily zejména k ukázkám programových konstrukcí, se kterými jste se seznamovali. Nicméně velice často budete tvořit programy, které budou mít desítky, stovky či ještě více řádek kódu. Pro lepší orientaci v rozsáhlejších programech můžete používat komentáře. Mohou však nastat situace, kdy zjistíte, že v programu používáte na více místech v podstatě stejný kód nebo že by pro přehlednost bylo vhodné vyčlenit určitou část kódu zvlášť a mít možnost ji v případě potřeby vykonat.

Také se může stát, že budete předpokládat, že časem bude potřeba daný kód znovu použít, a proto si jej vyčleníte rovnou do metody. Například budete v programu chtít od uživatele načíst známky studenta a později z nich v jednom případě vypočíst průměr, v jiném případě najít nejlepší známku nebo nejčastěji se vyskytující známku.

Pro tento případ by bylo vhodné mít někde připravenu část kódu, která zajistí načtení libovolného počtu známek studenta, a tuto část kódu mít možnost z více míst programu použít. Právě pro tyto případy existují tzv. metody. Každá metoda má svůj unikátní název a obsahuje zdrojový kód, který chcete vykonat. My už jsme metody používali, ale zatím jsme nevytvořili žádnou vlastní. Vzpomeňte si např. na volání metody Console.WriteLine nebo Convert.ToInt32.

3.1 Metody bez parametrů

S metodami se budete setkávat v podstatě neustále. Nejdříve se začnete učit tvořit metody, které pro svou činnost nepotřebují žádné vstupní parametry. Takovýmto metodám říkáme bezparametrické nebo metody bez parametrů. Pojďme si tedy nyní vytvořit naši první metodu, která vypíše na konzoli text „Ahoj“.

```
using System;

namespace MojeAplikace2
{
    Počet odkazů: 0
    class Program
    {
        Počet odkazů: 0
        static void Main(string[] args)
        {
            PrvniMetoda();
        }

        Počet odkazů: 1
        static void PrvniMetoda()
        {
            Console.WriteLine("Ahoj");
        }
    }
}
```

Obrázek 3.1: Metoda bez parametrů

Vytvořili jsme metodu nazvanou *PrvniMetoda*. Všimněte si toho, že metoda *PrvniMetoda* je umístěna v rámci složených závorek v třídě *Program*. Obdobně je definovaná metoda *Main*, která je vstupním bodem programu.

Za názvem metody jsou kulaté závorky a tělo metody je umístěno ve složených závorkách, obdobně jako jsme definovali tělo cyklu. Před názvem metody jsou umístěna dvě klíčová slova: *static* a *void*. Klíčového slova *static* si zatím nevšímejme, ale pouze si zapamatujme, že zatím jej musíme uvádět. Za klíčovým slovem *static* je klíčové slovo *void*. Toto označuje tzv. prázdný návratový typ.

K ZAPAMATOVÁNÍ

Každá metoda musí mít jasně určeno, zdali vrací nějakou hodnotu, či nikoliv. Například metoda *Console.WriteLine*, která vypisuje na konzoli, používá právě prázdný návratový typ *void*. Pokud chcete definovat, že metoda nic nevrací, pak použijete návratový typ *void*,

jako jsme to udělali u naší první metody. Pokud metoda vrací hodnotu, musíme určit, jaký datový typ metoda vrací.

Zatím jsme se naučili používat datové typy jako např. *int, double, string* a další. Metoda může vracet libovolný datový typ. Pokyn, že má metoda vrátit příslušnou hodnotu, respektive proměnnou, zapíšete pomocí klíčového slova *return*, za které uvedeme proměnnou nebo hodnotu, kterou chceme vrátit. Zkusme si tedy vytvořit program, který bude obsahovat metodu, která vrátí číslo 3 jako návratovou hodnotu a v programu si toto číslo uložíme do proměnné, abychom s ním v případě potřeby mohli dále pracovat.

```
using System;

namespace MojeAplikace2
{
    Počet odkazů: 0
    class Program
    {
        Počet odkazů: 0
        static void Main(string[] args)
        {
            int navratovaHodnota = DruhaMetoda();
            Console.WriteLine(navratovaHodnota);
        }

        Počet odkazů: 1
        static int DruhaMetoda()
        {
            return 2;
        }
    }
}
```

Obrázek 3.2: Metoda s návratovou hodnotou

Důležité je, že jakmile se vykoná příkaz return, vykonávání kódu metody končí. Pokud by byl uveden ještě nějaký příkaz po return, pak tento nebude vykonán.

3.2 Metody s parametry

Velice často se setkáte s tím, že metoda bude pro svou činnost vyžadovat nějaké vstupní informace. Začněme jednoduchou metodou, která bude mít za úkol sečíst dvě čísla a jejich součet vrátit. Tato dvě čísla předáme metodě pomocí parametrů. Program s takovouto metodou by mohl vypadat následujícím způsobem.

```
using System;

namespace MojeAplikace2
{
    Počet odkazů: 0
    class Program
    {
        Počet odkazů: 0
        static void Main(string[] args)
        {
            int navratovaHodnota = Soucet(2, 3);
            Console.WriteLine(navratovaHodnota);
        }

        Počet odkazů: 1
        static int Soucet(int a, int b)
        {
            int vysledek = a + b;
            return vysledek;
        }
    }
}
```

Obrázek 3.3: Metoda s parametry

V metodě *Main* předáme čísla 2 a 3 metodě *Soucet*, která provede výpočet a výsledek vrátí jako návratovou hodnotu, kterou si uložíme do proměnné *navratovaHodnota* a její hodnotu poté vypíšeme na obrazovku. Věnujme se nyní zejména definici parametrů metody. Metoda může mít v podstatě neomezeně parametrů nebo také žádný, jak jsme si ukazovali v podkapitole Metody bez parametrů.

V definici metody do závorek za její název můžeme definovat její parametry. U každého parametru musíme uvést jeho datový typ a název obdobně, jako když definujeme proměnnou. V rámci metody pracujeme s parametry metody naprosto stejně, jako bychom v metodě definovali proměnnou.

V metodě *Soucet* tedy pracujeme s proměnnými *a* a *b*. Pokud metoda vyžaduje pro svou činnost parametry, musíme jí je při jejím zavolání předat. Učiníme to tak, že do závorek za název volané metody zadáme proměnné nebo hodnoty, které chceme metodě předat, a to v tom pořadí, v jakém je definice metody očekává. Zároveň je nezbytné, aby předané parametry byly toho datového typu, který je uveden v definici metody. V případě metody součtu by nám záměna pořadí nevadila, ale kdyby metoda počítala například podíl, problém by už to byl.

Zkusme se ještě jednou podívat na tělo metody *Soucet*, kde nejprve definujeme takzvanou lokální proměnnou *vysledek*, která je typu *int*. Lokální proměnná znamená, že tato proměnná existuje pouze v rámci metody, ve které je definována. To znamená, že k této proměnné nemůžeme přistupovat z metody *Main* ani z žádné jiné metody. Stejně tak to znamená že můžeme definovat proměnnou se stejným názvem i v jiných metodách a nemusíme tak vymýšlet jiné názvy proměnných.

Použití lokálních proměnných jakožto dočasných úložišť pro různé mezivýpočty je v metodách poměrně obvyklé, ovšem v případě takto jednoduchého výpočtu jako je součet je možné zapsat vlastní výpočet přímo za klíčové slovo return a tím i tělo metod zkrátit. Zkusme si nyní analogicky vytvořit metodu pro součin dvou čísel, ale tentokrát již bez využití pomocné lokální proměnné *vysledek*.

```
using System;

namespace MojeAplikace2
{
    Počet odkazů: 0
    class Program
    {
        Počet odkazů: 0
        static void Main(string[] args)
        {
            int navratovaHodnota = Soucin(2, 3);
            Console.WriteLine(navratovaHodnota);
        }

        Počet odkazů: 1
        static int Soucin(int a, int b)
        {
            return a * b;
        }
    }
}
```

Obrázek 3.4: Metoda pro součin dvou čísel

Nyní si ukážeme ještě metodu pro výpočet podílu dvou celých čísel.

```
using System;

namespace MojeAplikace2
{
    Počet odkazů: 0
    class Program
    {
        Počet odkazů: 0
        static void Main(string[] args)
        {
            int navratovaHodnota = Podil(2, 3);
            Console.WriteLine(navratovaHodnota);
        }

        Počet odkazů: 1
        static int Podil(int a, int b)
        {
            return a / b;
        }
    }
}
```

Obrázek 3.5: Metoda pro podíl dvou čísel

Pokud výše uvedený program spustíme, výsledkem nebude 0.666…, jak bychom očekávali, ale program vypíše jako výsledek číslo nula. Důvodem je to, že návratový typ metody Podíl jsme definovali jako *int*, kdežto u podílu dvou celých čísel nemusí být výsledek vždy celočíselný. Ovšem ani když změníme návratový typ na *double*, nedostaneme správný výsledek. Dalším důvodem je totiž to, že u operátoru / jsou oba dva operandy (tedy proměnné *a* a *b*) celá čísla. V takovém případě je podíl vypočítán v oboru celých čísel, tj. desetinná část je odříznuta a zbude tedy nula. Abychom tomu předešli, musí být alespoň jeden z operandů reálného datového typu. Tedy buď změníme deklaraci parametru *a* nebo *b* na *double*, případně jeden z těchto parametrů při provádění operace podíl přetypujeme na *double*. Nyní si ukážeme, jak by upravený kód vypadal pro druhou možnost.

```
using System;

namespace MojeAplikace2
{
    Počet odkazů: 0
    class Program
    {
        Počet odkazů: 0
        static void Main(string[] args)
        {
            double navratovaHodnota = Podil(2, 3);
            Console.WriteLine(navratovaHodnota);
        }

        Počet odkazů: 1
        static double Podil(int a, int b)
        {
            return a / (double)b;
        }
    }
}
```

Obrázek 3.6: Upravená metoda pro podíl dvou čísel

3.3 Metody s parametry předávané odkazem

Doposud jsme vždy předávali metodám všechny parametry hodnotou, což znamená, že při předání parametru se v metodě vytvoří jeho kopie a když uvnitř metody změníme hodnotu takového parametru nedojde k přepsání původní proměnné v místě, kde byla metoda zavolána. Jedná se tedy o jakousi ochranu hodnot proměnných.

Ovšem v některých případech bychom naopak potřebovali, aby metoda mohla měnit hodnoty parametrů. Metody totiž můžou standardně vracet nejvýše jednu návratovou hodnotu. Řekněme, že bychom ale chtěli vytvořit metodu, která nám vypočítá obvod a obsah kruhu. Samozřejmě, že není problém vytvořit dvě samostatné metodu, jednu pro výpočet obvodu kruhu a druhou pro výpočet obsahu kruhu. My si ale ukážeme, jak k tomuto účelu lze využít parametry předávané odkazem. Podívejme se na názornou ukázku našeho příkladu.

```
using System;

namespace MojeAplikace2
{
    Počet odkazů: 0
    class Program
    {
        Počet odkazů: 0
        static void Main(string[] args)
        {
            double obsah = 0;
            double obvod = ObvodObsahKruhu(3, ref obsah);
            Console.WriteLine("Obvod: " + obvod + " obsah: " + obsah);
        }

        Počet odkazů: 1
        static double ObvodObsahKruhu(double r, ref double obsah)
        {
            obsah = Math.PI * Math.Pow(r, 2);
            return 2 * Math.PI * r;
        }
    }
}
```

Obrázek 3.7: Metoda s parametrem předaným odkazem

Metoda *ObvodObsahKruhu* počítá obvod i obsah kruhu, přičemž obvod kruhu vrací jako standardní návratovou hodnotu, kdežto k vrácení obsahu kruhu využívá parametru obsah, který je deklarován jako předaný odkazem pomocí klíčového slova *ref*. Důležitá je, že toto klíčové slovo *ref* musíme použít nejen při deklaraci metody, ale i při jejím volání.

Při výpočtu obvodu a obsahu kruhu jsme využili některých metod a konstant definovaných v knihovně *Math*. Konkrétně se jedná o konstantu *Math.PI*, která obsahuje přesnější hodnotu čísla π než jen 3,14. Dále jsme využili metodu *Math.Pow*, která počítá obecnou mocniny ve tvaru x^y. Z dalších často používaných metod můžeme ještě uvést metodu *Math.Sqrt*, která počítá druhou odmocninu. Jinak je v této knihovně celá řada dalších matematických funkcí, které odpovídají standardním matematickým funkcím. Stačí v kódu napsat Math a tečku a editor ve Visual Studiu vám prostřednictvím tzv. *IntelliSense* nabídne seznam dostupných funkcí.

OTÁZKY

1. Jaké klíčové slovo použijeme, pokud metoda nevrací žádnou návratovou hodnotu?
    a. void

   b. none
   c. empty
2. Který příkaz použijeme pro vrácení návratové hodnoty metody?
   a. return
   b. back
   c. equal
3. Pokud má metoda více parametrů, kterým symbolem oddělujeme jednotlivé parametry?
   a. Pomlčkou
   b. Středníkem
   c. Čárkou
4. Pokud má metoda definovaný celočíselný parametr a my tuto metodu zavoláme s reálným parametrem
   a. Parametr se převede automaticky na celé číslo
   b. Dojde k sémantické chybě
   c. Dojde k syntaktické chybě
5. Základní datové typy se standardně předávají
   a. Odkazem
   b. Hodnotou
   c. Nepředávají se
6. Pokud chceme definovat parametr metody jako předávaný odkazem, použijeme klíčové slovo
   a. refernce
   b. ref
   c. link
7. Co se stane, když deklarujeme dvě metody se stejným názvem?
   a. Dojde k syntaktické chybě
   b. Bude fungovat pouze metoda deklarovaná jako první
   c. Budou fungovat obě metody
8. Pokud chceme, aby metoda vrátila více než jednu návratovou hodnotu
   a. Použijeme k tomu parametry předávané odkazem
   b. Za příkazem return můžeme uvést více hodnot oddělených čárkou
   c. Metoda nemůže vracet více než jednu návratovou hodnotu
9. Co se stane, když deklarujeme metodu s návratovou hodnotou typu *int*, ale v těle metody neuvedeme příkaz *return*?
   a. Metoda se převede automaticky na metodu bez návratové hodnoty
   b. Dojde k syntaktické chybě
   c. Dojde k sémantické chybě
10. Proměnná deklarovaná uvnitř metody je přístupná
   a. V celém programu
   b. Pouze v metodách se stejným názvem, v níž je proměnná deklarovaná
   c. Pouze v metodě, ve které je deklarovaná

SHRNUTÍ KAPITOLY

V této kapitole jsme se naučili vytvářet metody a ukázali jsme si, jak je můžeme z programu opakovaně volat. Zároveň jsme si demonstrovali, jak můžeme modifikovat chování metody v závislosti na hodnotách parametrů. Ukázali jsme si rovněž, jak může metoda vracet návratovou hodnotu a s využitím parametrů předávaných odkazem jsme si předvedli, jak můžeme vracet více než jednu návratovou hodnotu.

ODPOVĚDI

1. a
2. a
3. c
4. c
5. b
6. b
7. c
8. a
9. b
10. c

4 ŘÍZENÍ BĚHU PROGRAMU

RYCHLÝ NÁHLED KAPITOLY

Nyní umíte vytvořit program, který postupně vykoná všechny příkazy, které jste zapsali. Představte si ale situaci, že budete chtít napsat program, který od uživatele načte dvě čísla a poté podle volby uživatele vypíše jejich součet nebo rozdíl. Jinými slovy budete chtít napsat takový program, který vykoná jednu sadu příkazů, pokud uživatel zvolí možnost součtu, a druhou sadu příkazů v případě, že uživatel zvolí možnost rozdílu. Toto se právě v této kapitole naučíte.

CÍLE KAPITOLY

Po prostudování této kapitoly budete umět:

- Rozdělit běh programu na více větví podle určitých podmínek.
- Jak pracovat s logickými výrazy.
- Vyhodnotit základní logické operátory.
- Poznat případy kdy nahradit příkaz *if* příkazem *switch*
- Nahradit jednoduchou podmínku ternárním operátorem.

KLÍČOVÁ SLOVA KAPITOLY

Podmínka, logický výraz, logická proměnná, ternární operátor.

4.1 Příkaz if

Jednou z nejdůležitějších konstrukcí při programování je podmínečný příkaz, tedy jak vykonat určitou část programu pouze tehdy, když platí nějaká podmínka. Za tímto účelem se v jazyce C# používá příkaz *if*, za který se do závorek zapíše podmínka, která určí, zdali se provede část programu, která je za příkazem *if* uvedena ve složených závorkách. Takto vypadá obecný zápis příkazu *if*.

```
if (logický výraz)
{
        Kód, který se vykoná, pokud je logický výraz vyhodnocen jako pravdivý
}
```

Podívejme se nyní na praktickou ukázku použití příkazu *if*.

```
static void Main(string[] args)
{
    int x = 3;
    int y = 4;

    if (x > y)
    {
        Console.WriteLine("Tento text nebude vypsán");
    }

    if (x <= y)
    {
        Console.WriteLine("Tento text bude vypsán");
    }
}
```

Obrázek 4.1: Příkaz if

Když se pozorněji podíváme na výše uvedený příklad, můžeme si všimnout, že obsahuje dvě opačné podmínky, protože když *x* není větší než *y*, potom logicky musí být *x* menší nebo rovno než *y*. Se situací, kdy při splnění podmínky chceme vykonat určitou část kódu a v opačném případě jinou část kódu, se budeme setkávat poměrně často. Proto lze příkaz *if* doplnit ještě o příkaz *else*, kterým určíme, že se má daná část programu vykonat, pokud není podmínka u příkazu *if* splněna. Příkaz *else* má následující syntaxi.

```
if (logický výraz)
{
        Kód, který se vykoná, pokud je logický výraz vyhodnocen jako pravdivý
}
else
{
        Kód, který se vykoná, pokud je logický výraz vyhodnocen jako nepravdivý
}
```

Výše uvedený příklad bychom tedy mohli upravit takto.

```
static void Main(string[] args)
{
    int x = 3;
    int y = 4;

    if (x > y)
    {
        Console.WriteLine("Tento text nebude vypsán");
    }
    else
    {
        Console.WriteLine("Tento text bude vypsán");
    }
}
```

Obrázek 4.2: Příkaz else

Tak jak jsme si příkazy *if* a else doposud definovali, můžeme pomocí nich rozdělit provádění programu do dvou větví. Někdy však potřebujeme rozdělit provádění do více než dvou větví. Toho lze dosáhnout vnořováním příkazů *if*. Drouhou možností podmínit i vykonání části programu definované příkazem *else*. K tomuto slouží příkaz *else if*. K příkazu *if* lze definovat libovolný počet příkazů *else if* a žádný nebo jeden příkaz *else*. Podívejte se na následující kompletní ukázku syntaxe popsaných příkazů.

```
if (logický výraz 1)
{
        Kód, který se vykoná, pokud je logický výraz 1 vyhodnocen jako pravdivý
}
else if (logický výraz 2)
{
        Kód, který se vykoná, pokud je logický výraz 1 vyhodnocen jako nepravdivý a zároveň je
        logický výraz 2 vyhodnocen jako pravdivý
}
else if (logický výraz 3)
{
        Kód, který se vykoná, pokud jsou logický výrazy 1 a 2 vyhodnoceny jako nepravdivé a zá-
        roveň je logický výraz 3 vyhodnocen jako pravdivý
}
...
...
...
else if (logický výraz N)
{
        Příkazů else if může být 0..N
}
```

```
else
{
        Pokud je definován příkaz else, vykoná se tento kód právě tehdy, když jsou všechny výše
        uvedené logické výrazy vyhodnoceny jako nepravdivé
}
```

Nyní si ukážeme použití výše uvedené konstrukce na praktickém příkladu, kdy budeme mít v celočíselné proměnné x uložené číslo od 1 do 3 a naším úkolem bude na obrazovku vypsat slovní hodnotu tohoto číslo, tj. jedna, dvě a tři. Pokud bude číslo menší než jedna nebo větší než tři, vypíšeme informaci, že číslo je mimo rozsah.

```
static void Main(string[] args)
{
    int x = 2;

    if (x == 1)
    {
        Console.WriteLine("Bylo zadáno číslo jedna");
    }
    else if (x == 2)
    {
        Console.WriteLine("Bylo zadáno číslo dva");
    }
    else if (x == 3)
    {
        Console.WriteLine("Bylo zadáno číslo tři");
    }
    else
    {
        Console.WriteLine("Zadané číslo je mimo rozsah");
    }
}
```

Obrázek 4.3: Příkaz else if

K ZAPAMATOVÁNÍ

Příkaz *else if* nám umožňuje vyhnout se vnořeným podmínkám.

4.2 Příkaz switch

Přehlednost a čitelnost kódu velmi důležitým aspektem jeho použitelnosti a rozšiřitelnosti, proto si ukážeme se ještě jeden příkaz, a to příkaz *switch*, který vám umožní elegantně nahradit několik podmínek zapsaných pomocí příkazů *if* a *else*. Příkaz switch má následující syntaxi.

```
switch (proměnná nebo výraz)
{
        case hodnota1:
                Příkazy provedené, pokud výraz odpovídá hodnotě 1
                break;


        case hodnota2:
                Příkazy provedené, pokud výraz odpovídá hodnotě 2
                break;
        ...
        ...
        ...
        case hodnotaN:
                Příkazy provedené, pokud výraz odpovídá hodnotě 3
                break;
        default: (nepovinný – není nutné jej vždy uvést)
                Příkazy provedené, pokud výraz neodpovídal ani jedné hodnotě u case
                break;
}
```

Nejdříve určíte, který výraz bude vyhodnocen. Tento je vyhodnocen pouze jednou a poté se provedou příkazy za klíčovým slovem *case*, jehož hodnota odpovídá výrazu. Pokud ani jedna hodnota neodpovídá, jsou provedeny příkazy za klíčovým slovem *default*. Příkaz default je volitelný, není nutné vždy uvést. Každý sled příkazů za příkazem *case* či *default* musí končit klíčovým slovem *break*.

Pojďme si jej vyzkoušet na předchozím příkladu, kdy jsme převáděli hodnotu čísla na jeho slovní popis.

```
static void Main(string[] args)
{
    int x = 2;

    switch (x)
    {
        case 1:
            Console.WriteLine("Bylo zadáno číslo jedna");
            break;
        case 2:
            Console.WriteLine("Bylo zadáno číslo dva");
            break;
        case 3:
            Console.WriteLine("Bylo zadáno číslo tři");
            break;
        default:
            Console.WriteLine("Zadané číslo je mimo rozsah");
            break;
    }
}
```

Obrázek 4.4: Příkaz switch

Sami můžete porovnat, který z obou způsobů je přehlednější. Zkusme si ještě ukázat jen příklad na příkaz *switch*, kdy budeme mít v proměnné operace uložený symbol příslušné operace. Našim cílem bude vypsat na obrazovku slovní popis zadané operace, přičemž u operací sčítání a odčítání budeme chtít navíc mít možnost zadat operaci nejen pomocí symbolu operace ale i podle počátečního písmena, tj. s-sčítání a o-odčítání.

```
static void Main(string[] args)
{
    string operace = "+";
    switch (operace)
    {
        case "+":
        case "s":
            Console.WriteLine("Bylo zadána operace sčítání");
            break;
        case "o":
        case "-":
            Console.WriteLine("Bylo zadána operace odčítání");
            break;
        case "*":
            Console.WriteLine("Bylo zadána operace násobení");
            break;
        case "/":
            Console.WriteLine("Bylo zadána operace dělení");
            break;
        default:
            Console.WriteLine("Byla zadána neplatná operace");
            break;
    }
}
```

Obrázek 4.5: Příkaz switch s více možnostmi case

Jak vidíme zadání více možností pro určitou část kódu je velice jednoduché, stačí nad sebou uvést výčet několika návěští *case*. Při použití příkazu *if* bychom již museli používat složené podmínky. Dalším benefitem u příkazu *switch* je automatická kontrola duplicity hodnot v návěštích *case*. Představme si, že bychom omylem pro operaci násobení uvedli *case „+“*. V takovém případě by nám překladač oznámil syntaktickou chybu, že se v příkazu switch nachází duplicitní hodnoty. Takto takovou chyby najdeme velmi snadno, naopak kdybychom stejný příklad realizovali pomocí příkazu *if* a udělali stejnou chybu, překladač by nám žádnou syntaktickou chybu neoznámil a museli bychom ji najít sami pomocí ladění.

K ZAPAMATOVÁNÍ

Velkou výhodou příkazu *switch* je syntaktická kontrola duplicity hodnot.

4.3 Ternární operátor

Již jsme si ukázali dva příkazy pro větvení programu, pomocí níž jsme schopni napsat libovolnou podmínku. V případě jednoduchých podmínek, kdy nás zajímají pouze dvě možnosti, lze takovouto podmínku nahradit pomocí ternárního operátoru. Dříve než se k němu dostaneme napíšeme si nejprve metodu, která nám maximum ze dvou celých čísel zadaných jako parametry.

```
static void Main(string[] args)
{
    int vysledek = Max(2, 3);
    Console.WriteLine(vysledek);
}

Počet odkazů: 1
static int Max(int x, int y)
{
    if (x > y)
    {
        return x;
    }
    else
    {
        return y;
    }
}
```

Obrázek 4.6: Maximum z dvou čísel pomocí příkazu if

Výše uvedená metoda je velice jednoduchá, pouze otestuje, zda je první parametr větší než druhý a podle toho vrátí jako návratovou hodnotu první nebo druhý parametr. Jedinou nevýhodou je, že tělo metody zabírá relativně hodně místa. Nyní si zkusíme stejnou metodu přepsat pomocí ternárního operátoru.

```
static void Main(string[] args)
{
    int vysledek = Max(2, 3);
    Console.WriteLine(vysledek);
}

Počet odkazů: 1
static int Max(int x, int y)
{
    return x > y ? x : y;
}
```

Obrázek 4.7: Maximum ze dvou čísel pomocí ternárního operátoru

Jak můžeme vidět, zápis podmínky je mnohem stručnější. Všimněte si, že na rozdíl od způsobu pomocí příkazu *if*, je u použití ternárního operátoru pouze jeden příkaz *return*. Je tomu tak proto, že ternární operátor vrací jako výsledek přímo hodnotu, podobně jako jiné operátory, jen na rozdíl od nich má ternární operátor tři operandy. Uveďme si opět obecný zápis ternárního operátoru.

logický_výraz ? hodnota1 : hodnota2

Ternární operátor je složen ze tří částí oddělených symboly „?" a „:". První část uvedená před otazníkem je logický výraz, v případě že tento logický výraz je pravdivý, bude výsledkem ternárního operátoru hodnota uvedená bezprostředně za symbolem otazníku. Pokud naopak logický výraz bude nepravdivý, bude výsledkem ternárního operátoru hodnota uvedená bezprostředně za symbolem dvojtečky.

K ZAPAMATOVÁNÍ

Ternární operátor lze vždy nahradit použitím podmínky *if else*, ale ternární operátor nám zjednodušuje zápis kódu.

4.4 Logické výrazy a operátory

Prozatím jsme si ukázali větvení programu pomocí velice jednoduchých podmínek, často však budeme potřebovat vyjádřit i mnohem složitější podmínky. Nejprve se podrobněji seznámíme s datovým typem *Boolean*, který může nabývat pouze dvou hodnot – *true*

a *false* neboli česky pravda – nepravda. Tento typ deklarujeme pomocí klíčového slova *bool*. Ukažme si, jak definovat proměnnou, která bude logického datového typu *bool*.

```
bool promenna1 = true;
bool promenna2 = false;
```

S logickým datovým typem úzce souvisí pojem logický výraz. Jde o výraz, jehož výsledkem je právě hodnota *true* nebo *false* výše zmíněného logického datového typu *bool*. Logický výraz musí splňovat pravidlo, že je jednoznačně vyhodnotitelné, zdali je pravdivý. Příkladem logického výrazu, u nějž můžeme prohlásit, že je pravdivý, může být výraz 3 > 2. Naopak nepravdivým výrazem může být např. výraz 2 > 10. Pojďme si nyní představit nejčastěji používané logické výrazy, které budeme používat.

Tabulka 4.1: Logické výrazy

Výraz	Význam
x == y	*x* je rovno *y*
x < y	*x* je menší než *y*
x > y	*x* je větší než *y*
x <= y	*x* je menší nebo rovno *y*
x >= y	*x* je větší nebo rovno *y*
x != y	*x* není rovno *y*

Velice často se setkáte s tím, že budete muset zapsat komplexnější logický výraz. Představte si situaci, kdy bude chtít zapsat výraz, kterým budete moci vyhodnotit, zdali je zadaný věk nějakého člověka přípustný. Podmínku si zjednodušme tak, že věk musí být nezáporný a zároveň je zadaný věk menší než nějaký maximálně přípustný věk, např. 150. Abychom mohli tento výraz zapsat, musíme použít logický operátor „a zároveň“, který v jazyce C# zapíšeme jako &&. Kromě operátoru && existují i operátory || (nebo) a ! (negace). Podívejte se na následující přehled logických operátorů, kde je uvedeno jejich vyhodnocení pro jednotlivé logické hodnoty.

Tabulka 4.2: Logické operátory

x	*y*	*!x*	*!y*	*x && y*	*x || y*
true	*true*	*false*	*false*	*true*	*true*
true	*false*	*false*	*true*	*false*	*true*
false	*true*	*true*	*false*	*false*	*true*
false	*false*	*true*	*true*	*false*	*false*

Zkusme si nyní zapsat výše uvedený příklad pro vyhodnocení přípustného věku pomocí podmínky *if*.

```
static void Main(string[] args)
{
    int vek = 25;
    if (vek >= 0 && vek < 150)
    {
        Console.WriteLine("Zadaný věk je přípustný");
    }
    else
    {
        Console.WriteLine("Zadaný věk není přípustny");
    }
}
```

Obrázek 4.8: Podmínka s logickým operátorem

OTÁZKY

1. Blok kódu, který je uveden bezprostředně za příkazem *if*, se vykoná, pokud je výraz v podmínce
    a. Roven hodnotě false
    b. Roven hodnotě true
    c. Uveden v hranatých závorkách
2. Podmínku u příkazu *if* zapisujeme do závorek
    a. Složených
    b. Hranatých
    c. Kulatých
3. Pokud chceme v podmínce vyjádřit „logické nebo“, použijeme k tomu zápis
    a. ||
    b. OR
    c. ++
4. Pokud chceme v podmínce vyjádřit „logické a“, použijeme k tomu zápis
    a. --
    b. AND
    c. &&
5. Pokud chceme v podmínce vyjádřit „logickou negaci“, použijeme k tomu zápis
    a. NOT
    b. !
    c. !!
6. Příkaz switch se používá zejména
    a. Pokud testujeme jednu proměnnou na více hodnot

   b. Pokud se nám nelíbí příkaz *if*
   c. Když chceme nahradit cyklus *for*
7. Kolik operandů má ternární operátor?
   a. Jeden
   b. Dva
   c. Tři
8. Který příkaz nemůžeme použít u podmínky *if*?
   a. else
   b. case
   c. else if
9. Které klíčové slovo použijeme v příkazu *switch* pro možnost, která nevyhovuje žádné z uvedených možností?
   a. default
   b. else
   c. other
10. Kterým příkazem ukončujeme v příkazu switch každou větev programu?
   a. end
   b. stop
   c. break

SHRNUTÍ KAPITOLY

V této kapitole jsme se seznámili se základními příkazy pro podmínečné vykonávání části programu v závislosti na určitých podmínkách. Pomocí příkazu if můžeme specifikovat libovolnou podmínku, oproti tomu příkaz switch je specializovaný pro testování jedné proměnné na více hodnot. Dále jsme si ukázali, jak můžeme nahradit jednoduchou podmínku pomocí ternárního výrazu a tím tak podstatně zkrátit příslušný kód. Nakonec jsem se vysvětlili základy práce s logickými výrazy a operátory.

ODPOVĚDI

1. b
2. c
3. a
4. c
5. b
6. a
7. c
8. b

9. a
10. c

5 CYKLY A POLE

RYCHLÝ NÁHLED KAPITOLY

Velmi často se setkáte s tím, že budete potřebovat, aby program určité příkazy vykonal opakovaně. Představte si situaci, kdy budete chtít vytvořit program, který uživateli umožní sečíst libovolné množství čísel. S obdobnou úlohou byste si již poradili za předpokladu, že by bylo pevně dáno, kolik čísel budete sčítat. Pokud počet čísel neznáte a chcete, aby program fungoval tak, že se po každém zadaném čísle zeptá, zdali chce uživatel zadat další, musíte se naučit další konstrukce jazyka C# umožňující určitou sadu příkazů vykonat opakovaně. Tyto konstrukce nazýváme cykly. Zároveň se naučíte používat nový datový typ pole, který vám umožní si do proměnné uložit v podstatě libovolné množství dat.

CÍLE KAPITOLY

Po prostudování této kapitoly budete umět:

- Vyjmenovat základní druhy cyklů a způsob jejich použití.
- Používat cykly pro opakované vykonání kódu.
- Vyřešit situaci, kdy potřebujete, aby cyklus proběhl za všech okolností alespoň jednou
- Deklarovat pole.
- Vypsat prvky pole.
- Provádět různé operace s prvky pole.
- Správně indexovat prvky pole.

KLÍČOVÁ SLOVA KAPITOLY

Cyklus, iterace, iterační proměnná, pole, index.

5.1 Cyklus for

Tento cyklus bude užitečný zejména ve chvíli, kdy budete potřebovat provést předem známý počet opakování cyklu. Cyklus *for* má následující syntaxi.

```
for (inicializační výraz; ukončující logický výraz; změna řídící proměnné)
{
        // Příkazy v těle cyklu
}
```

V rámci tohoto cyklu se používá tzv. řídicí proměnná, kterou použijete pro řízení počtu opakování cyklu *for*. Tuto proměnnou nejdříve inicializujete na výchozí hodnotu a po každém průběhu cyklu její hodnotu změníte. Cyklus se opakuje, dokud ukončující logický výraz vrací hodnotu *true*. Podívejte se na následující program, který na příkazový řádek vypíše čísla od nuly do desítky.

```
static void Main(string[] args)
{
    for (int i = 0; i <= 10; i++)
    {
        Console.WriteLine(i);
    }
}
```

Obrázek 5.1: Cyklus for

Nejdříve se v první části definice cyklu vytvoří proměnná *i* s výchozí hodnotou 0, která je typu *int*. Cyklus se bude opakovat, dokud je *i* <= 10, což je dáno druhou částí definice cyklu. Po každém průběhu cyklu se hodnota proměnné *i* zvýší o jedničku, což nám zajišťuje poslední část definice cyklu *for*. Všimněte si, že pro zvýšení hodnoty proměnné *i* o 1 využíváme zkrácený zápis *i++*, úplně stejného výsledku bychom dosáhli použitím zápisu *i+=1*, nebo *i = i + 1*. Zkusme si nyní vytvořit metodu s celočíselným parametrem *n*, která vypočítá součet řady celých čísel od 1 do zadaného parametru *n*. Potom tuto metodu zavoláme z metody *main* s parametrem 10.

```
class Program
{
    Počet odkazů: 0
    static void Main(string[] args)
    {
        int soucet = SoucetRady(10);
        Console.WriteLine(soucet);
    }

    Počet odkazů: 1
    static int SoucetRady(int n)
    {
        int suma = 0;
        for (int i = 1; i <= n; i++)
        {
            suma = suma + i;
        }
        return suma;
    }
}
```

Obrázek 5.2: Metoda součet řady

Jak vidíme metoda *SoucetRady* využívá pomocné proměnné suma, do které v cyklu *for* postupně přičítá hodnoty iterační proměnné *i*. Na konci cyklu potom metoda vrací hodnotu proměnné suma jako návratovou hodnotu. Po spuštění tohoto programu bude výsledkem číslo 55.

K ZAPAMATOVÁNÍ

Cyklus *for* je vhodné použít zejména v situacích, kdy předem známe počet opakování.

5.2 Cyklus while

Cyklus *while* funguje tak, že definujete výraz, který se před každým průběhem cyklu vyhodnotí, a pokud je vyhodnocen jako pravdivý, příkazy uvedené v těle cyklu se vykonají.

Průběhem cyklu tedy rozumíme právě jedno vykonání příkazů v těle cyklu. Cyklus *while* má následující syntaxi:

```
while (logický výraz)
{
        // Příkazy v těle cyklu
}
```

Po skončení jednoho průběhu cyklu se opět vyhodnotí logický výraz. Pokud je vyhodnocen jako pravdivý, příkazy v těle cyklu se vykonají znovu. Snadno nahlédneme, že pokud by byl výraz vyhodnocen vždy jako pravdivý, bude se cyklus vykonávat donekonečna. Na toto je potřeba dávat pozor, protože nekonečný cyklus způsobí, že program nikdy neskončí. Pojďme si cyklus *while* vyzkoušet na následujícím příkladu. Zkusme si nyní ukázat použití cyklu *while* na stejném příkladu, který jsme si ukazovali u cyklu *for*, tedy výpis čísel od 0 do 10.

```
static void Main(string[] args)
{
    int i = 0;
    while (i <= 10)
    {
        Console.WriteLine(i);
        i++;
    }
}
```

Obrázek 5.3: Cyklus while

Všimněte si, že zde u tohoto cyklu musíme nejprve definovat iterační proměnnou *i*, v samotném cyklu *while* pouze testujeme, zda je hodnota i stále menší nebo rovna 10. Rovněž tak zvýšení hodnoty proměnné *i* neprovádíme v definici cyklu, ale až v těle cyklu.

Nyní si ještě zkusíme přepsat metodu pro součet řady s využitím cyklu *while*.

```
class Program
{
    Počet odkazů: 0
    static void Main(string[] args)
    {
        int soucet = SoucetRady(10);
        Console.WriteLine(soucet);
    }

    Počet odkazů: 1
    static int SoucetRady(int n)
    {
        int suma = 0;
        int i = 1;
        while (i <= n)
        {
            suma += i;
            i++;
        }
        return suma;
    }
}
```

Obrázek 5.4: Cyklus while - součet řady

Jak můžeme vidět, na začátku metody definujeme nejen pomocnou proměnnou suma, ale i iterační proměnnou *i*. V těle cyklu potom přičítáme k proměnné suma aktuální hodnotu iterační proměnné *i*, zde jsme jen využili zkrácený zápis *suma+=i*, což je ekvivalentní zápisu *suma=suma+i*.

5.3 Cyklus do-while

Jak již název tohoto cyklu napovídá, má mnoho společného s cyklem *while*. Jediný rozdíl mezi cykly *do-while* a *while* je ten, že u cyklu *do-while* se výraz určující, zdali se cyklus bude opakovat, zapisuje až na konec cyklu. Cyklus *do-while* se tedy vždy vykoná nejméně jednou. Syntaxe tohoto cyklu je následující:

```
do
{
        // Příkazy v těle cyklu
} while (logický výraz)
```

Zkusme si nyní přepsat oba příklady na výpis čísel od 0 do 10 a součet řady pomocí cyklu *do-while*.

```
static void Main(string[] args)
{
    int i = 0;
    do
    {
        Console.WriteLine(i);
        i++;
    } while (i <= 10);
}
```

Obrázek 5.5: Cyklus do-while

```
class Program
{
    Počet odkazů: 0
    static void Main(string[] args)
    {
        int soucet = SoucetRady(10);
        Console.WriteLine(soucet);
    }

    Počet odkazů: 1
    static int SoucetRady(int n)
    {
        int suma = 0;
        int i = 1;
        do
        {
            suma += i;
            i++;
        } while (i <= n);
        return suma;
    }
}
```

Obrázek 5.6: Cyklus do-while pro součet řady

Můžeme vidět, že cyklus *do-while* je velmi podobný cyklu *while*, musíme však dávat pozor na to, že cyklus *do-while* proběhne vždy alespoň jednou, tedy i kdybychom metodu *SoucetRady* zavolali s parametrem 0, výsledkem bude číslo 1. Toto chování najde uplatnění zejména při testování vstupu od uživatele, zda chceme například pokračovat.

K ZAPAMATOVÁNÍ

Cyklus *do-while* se vykoná vždy alespoň jednou bez ohledu na ukončující podmínku.

5.4 Pole

Představte si, že si chcete uložit nějaké údaje o více prvcích. Např. chcete v paměti uchovávat 10 čísel, políčka šachovnice nebo jména 50 uživatelů. Asi vám dojde, že v programování bude nějaká lepší cesta než začít deklarovat proměnné uzivatel1, uzivatel2, ... až uzivatel50. Nehledě na to, že jich může být třeba 1000.

Pokud potřebujeme uchovávat větší množství proměnných stejného typu, tento problém nám řeší pole. Můžeme si ho představit jako řadu přihrádek, kde v každé máme uložený jeden prvek. Přihrádky jsou očíslované tzv. indexy, jen je třeba si dávat pozor na to, že první má index 0.

Programovací jazyky se velmi liší v tom, jak s polem pracují. V některých jazycích (zejména starších, kompilovaných) nebylo možné za běhu programu vytvořit pole s dynamickou velikostí (např. mu dát velikost dle nějaké proměnné). Pole se muselo deklarovat s konstantní velikostí přímo ve zdrojovém kódu. Naopak některé interpretované jazyky umožňují nejen deklarovat pole s libovolnou velikostí, ale dokonce tuto velikost na již existujícím poli měnit (např. PHP). V jazyce C# můžeme pole definovat s velikostí, kterou dynamicky zadáme až za běhu programu, ale velikost existujícího pole modifikovat nemůžeme.

Nyní si ukážeme deklaraci různých polí na příkladu.

```
static void Main(string[] args)
{
    int[] poleCelychCisel = new int[5];
    double[] poleRealnychCisle = new double[5];
    string[] poleRetezcu = new string[5];
}
```

Obrázek 5.7: Deklarace pole

Všimněte si zejména toho, že za datovým typem *int*, *double* a *string* jsou uvedeny hranaté závorky, tím označujeme, že jde o pole daného datového typu. Dále si všimněte klíčového slova *new*, které je zapotřebí při vytváření nového pole. Velikost pole musíme zapsat také do hranatých závorek, jak je uvedeno v příkladu výše. Nyní se naučíte, jak do pole uložit hodnoty a jak je z pole přečíst a vypsat na obrazovku.

```
static void Main(string[] args)
{
    int[] cisla = new int[5];
    cisla[0] = 5;
    cisla[1] = 7;
    cisla[2] = 1;
    cisla[3] = 4;
    cisla[4] = 2;

    //A nyní čísla vypíšeme na obrazovku
    Console.WriteLine(cisla[0]);
    Console.WriteLine(cisla[1]);
    Console.WriteLine(cisla[2]);
    Console.WriteLine(cisla[3]);
    Console.WriteLine(cisla[4]);
}
```

Obrázek 5.8: Práce s polem

K ZAPAMATOVÁNÍ

První prvek pole má index roven nule, index posledního prvku je proto roven hodnotě o jedna menší, než je počet prvků pole.

Pokud známe hodnoty prvků pole v době psaní programu, existuje i jednodušší inicializace pole a nastavení jeho prvků. V tomto případě nemusíme vytvářet instanci pole, ale pole vytvoříme tak, že do množinových závorek uvedeme výčet všech jeho prvků. Zároveň si ukážeme, jak můžeme vypsat prvky pole pomocí cyklu.

```
static void Main(string[] args)
{
    int[] cisla = { 5, 7, 1, 4, 2 };

    //A nyní čísla vypíšeme na obrazovku pomocí cyklu
    for (int i = 0; i < cisla.Length; i++)
    {
        Console.WriteLine(cisla[i]);
    }
}
```

Obrázek 5.9: Deklarace pole výčtem prvků

Ve výčtu existujících druhů cyklů jsme záměrně opomněli ještě jeden cyklus *foreach*. Tento cyklus je specifický tím, že je určen zejména pro procházení prvků polí a obdobných datových typů. Tento cyklus funguje tak, že postupně projde jednotlivé prvky pole a umožní s nimi pracovat v těle cyklu. Důležité je, že cyklus projde vždy všechny prvky pole a umožní s každým jednotlivým prvkem pracovat v těle cyklu. Ukažme si syntaxi tohoto cyklu.

```
foreach (promennaProJedenPrvek in poleKtereProchazime)
{
        // Příkazy v těle cyklu
}
```

Použijeme tedy klíčové slovo *foreach*. V závorce následuje definice proměnné, která bude v každém běhu cyklu obsahovat právě jeden prvek z pole. Za touto proměnnou následuje klíčové slovo in a za ním proměnná obsahující pole, jehož prvky chceme procházet. Podívejme se na následující ukázku. V programu si připravíme jedno pole celých čísel a druhé pole řetězců a pomocí cyklu *foreach* vypíšeme jejich prvky na konzoli. Všimněte si, že proměnná, kterou v rámci cyklu *foreach* definujeme, musí být stejného datového typu jako prvky v příslušném poli.

```
static void Main(string[] args)
{
    //Nejprve deklarujeme pole čísel
    int[] cisla = { 5, 7, 1, 4, 2 };
    //A nyní čísla vypíšeme na obrazovku pomocí cyklu
    foreach (int prvek in cisla)
    {
        Console.WriteLine(prvek);
    }

    //Následně deklarujeme pole řetězců
    string[] retezce = { "alfa", "beta", "gama" };
    //A nyní řetězce vypíšeme na obrazovku pomocí cyklu
    foreach (string prvek in retezce)
    {
        Console.WriteLine(prvek);
    }
}
```

Obrázek 5.10: Cyklus foreach

V případě, že se rozhodujeme, který z cyklů použít, můžeme použít pravidlo, že pokud potřebujete projít všechny prvky pole a zároveň nepotřebujete mít informaci o pořadí aktuálně zpracovávaného prvku, je vhodnější použít cyklus *foreach*. Pokud potřebujete projít např. prvních deset prvků pole a zároveň mít k dispozici informaci o pořadí aktuálně zpracovávaného prvku, je vhodnější cyklus *for*.

K ZAPAMATOVÁNÍ

Cyklus *foreach* nemá ukončující podmínku, vždy projde všechny prvky pole.

OTÁZKY

1. Který cyklus použijeme, pokud chceme, aby se provedl alespoň jednou?
    a. for
    b. while
    c. do-while
2. Který cyklus nepoužívá ukončující podmínku?
    a. foreach

   b. for
   c. while
3. První prvek pole má hodnotu indexu rovnu
   a. 1
   b. 0
   c. U deklarace pole můžeme definovat hodnotu počátečního indexu pole
4. Co znamená, když v cyklu *for* napíšeme *i*++ ?
   a. Hodnota proměnné *i* se zvýší o 1
   b. Hodnota proměnné *i* se zvýší o 2
   c. Jedná se o syntaktickou chybu
5. Cyklus *while* se bude provádět, dokud je podmínka
   a. Pravdivá
   b. Nepravdivá
   c. Cyklus *while* proběhne vždy pouze jednou
6. Jednotlivé části cyklu *for* oddělujeme
   a. Čárkou
   b. Středníkem
   c. Mezerou
7. Které klíčové slovo používáme v cyklu *foreach*?
   a. in
   b. inside
   c. with
8. Je-li *cisla* proměnná typu pole celých čísel, jak zjistíme maximální počet čísel, které můžeme do tohoto pole uložit?
   a. cisla.Size
   b. cisla.Width
   c. cisla.Length
9. Máme-li cyklus *for* s iterační proměnnou *i* s výchozí hodnotou 1, co uvedeme do poslední částí cyklu, pokud chceme projít pouze liché hodnoty proměnné *i*?
   a. i++
   b. i++2
   c. i+=2
10. Máme-li deklarované pole celých čísel *cisla* o kapacitě 5 prvků a napíšeme příkaz *cisla*[5]=3, co se stane?
   a. Poslednímu prvku se přiřadí hodnota 3
   b. Program se nespustí
   c. Program se spustí, ale skončí chybou

SHRNUTÍ KAPITOLY

V této kapitole jsme si ukázali, jak opakovat vykonání určité části kódu. U každého cyklu jsme si uvedli základní charakteristiky a srovnali z jinými druhy cyklů. Dále jsme si ukázali nový datový typ pole, které se používá pro uložení více hodnot stejného typu, ke kterým můžeme přistupovat pomocí indexu. Nakonec jsme se seznámili s cyklem *foreach*, který je specializovaný pro procházení polí.

ODPOVĚDI

1. c
2. a
3. b
4. a
5. a
6. b
7. a
8. c
9. a
10. c

6 KOMUNIKACE PROGRAMU S OKOLÍM

RYCHLÝ NÁHLED KAPITOLY

Program, který bude dělat pořád totéž, nebude zřejmě úplně užitečný. Každý programovací jazyk proto nabízí možnosti, jak komunikovat s okolím a na základě toho měnit své chování. Zde si ukážeme, jaké jsou základní možnosti načtení vstupu od uživatele a jak potom získat výstupy z programu. Seznámíme se zejména s formátovaným výstupem na obrazovku, načítáním vstupu od uživatele z klávesnice, ukážeme si základy práce se soubory a nakonec si ukážeme, jak jednoduše můžeme předávat parametry na příkazovém řádku.

CÍLE KAPITOLY

Po prostudování této kapitoly budete umět:

- Zobrazit výsledek programu na obrazovku.
- Zobrazený výstup vhodně zformátovat.
- Načíst a zpracovat vstup od uživatele z klávesnice.
- Vytvořit soubor a zapsat do něho text.
- Načíst data z existujícího souboru.
- Ověřit, zda soubor existuje na disku.
- Používat parametry na příkazovém řádku.

KLÍČOVÁ SLOVA KAPITOLY

Vstup, výstup, konzole, formát, soubor, parametr.

6.1 Výstup na obrazovku

Základní příkaz pro výpis na obrazovku *Console.WriteLine* jsme si už představili, ovšem doposud jsme ho používali jen pro výpis hodnot proměnných nebo řetězců. Mnohdy však potřebujeme textový výstup vhodně naformátovat. Ukážeme si nyní příklad, jak můžeme formátovat proměnné vypisované na obrazovku pomocí operátoru „+“.

```
static void Main(string[] args)
{
    int vek = 25;
    double hmotnost = 75.5;
    string jmeno = "Pavel";
    Console.WriteLine("Jmenuji se " + jmeno + ", je mi " + vek + " let a vážím " + hmotnost + "kg.");
}
```

Obrázek 6.1: Formátovaný výpis na obrazovku

Výše uvedený příklad vypíše na obrazovku text „Jmenuji se Pavel, je mi 25 let a vážím 75,5kg.“ Další možností je využití parametrů zadaných v příkazu *Console.WriteLine* pomocí složených závorek. Zkusme si názorně přepsat předchozí příklad tímto způsobem.

```
static void Main(string[] args)
{
    int vek = 25;
    double hmotnost = 75.5;
    string jmeno = "Pavel";
    Console.WriteLine("Jmenuji se {0}, je mi {1} let a vážím {2}kg.", jmeno, vek, hmotnost);
}
```

Obrázek 6.2: Formátovaný výpis na obrazovku s parametry

Na první pohled jsou oba způsoby zápisu naprosto ekvivalentní, druhý způsob však umožňuje mnohem širší možnosti formátování a to tak, že za číslo parametru ve složených závorkách uvedeme znak dvojtečku a za ní takzvaný formátovací řetězec. Řekněme, že bychom chtěli předchozí příklad upravit tak, aby hmotnost byla zaokrouhlena na celé číslo. Toho docílíme tak, že u posledního parametru nastavíme počet desetinných míst na nulu, tj. {2:0.}. Celý příklad bude vypadat takto.

```
static void Main(string[] args)
{
    int vek = 25;
    double hmotnost = 75.5;
    string jmeno = "Pavel";
    Console.WriteLine("Jmenuji se {0}, je mi {1} let a vážím {2:0.}kg.", jmeno, vek, hmotnost);
}
```

Obrázek 6.3: Výpis na obrazovku s parametry a formátovacím řetězcem

Formátovacích řetězců existuje celá řada, následující tabulka shrnuje často používané formátovací řetězce včetně příkladů s konkrétními hodnotami.

Tabulka 6.1: Formátovací řetězce

Specifikátor	Popis	Formát	Vstup	Výstup
0	Vyplňuje nulami	{0:00.0000}	3.14	03.1400
#	Rozklad na cifry	{0:(#).##}	3.14	(3).14
.	Zaokrouhlení	{0:0.0}	3.14	3.1
,	Oddělovač tisíců	{0:0,0}	40000	40 000
%	Procento	{0:0%}	3.14	314%
e	Exponent	{0:00e+0}	3.14	31e-1
c	Měna	{0:c}	40	40.00 Kč
x	Hexadecimální hodnota	{0:x}	200	c8

K ZAPAMATOVÁNÍ

Pomocí formátovacích řetězců můžeme snadno ovlivnit, jak bude hodnota proměnné zobrazena na obrazovce.

6.2 Vstup od uživatele

Nedílnou součástí řady programů je získávání vstupu od uživatele. Jinými slovy programy mnohdy pro svou činnost potřebují informace, které jim uživatel v průběhu práce s programem zadá. Např. program, který by měl za úkol sečíst dvě čísla, by nejdříve uživatele vyzval k jejich zadání a následně by vypsal jejich součet na konzoli. Nyní se podíváme, jak získat informace od uživatele prostřednictvím konzole. Využijeme k tomu metodu *Console.ReadLine*, která nám vrátí načtený vstup od uživatele jako návratovou hodnotu typu *string*. Pojďme si nyní demonstrovat tuto metodu na příkladu, který od uživatele načte jeho jméno a poté jej použije při výpisu věty na konzoli.

```
static void Main(string[] args)
{
    Console.WriteLine("Zadejte jméno:");
    string jmeno = Console.ReadLine();
    Console.WriteLine("Zadané jméno bylo " + jmeno);
}
```

Obrázek 6.4: Načtení řetězce

Nyní již umíme načíst řetězec od uživatele a uložit si jej do proměnné. Nicméně v řadě případů budeme potřebovat získat od uživatele např. celé nebo desetinné číslo a dále s ním pracovat. Řekněme, že bychom měli napsat program, který od uživatele načte dvě reálná

čísla a vypíše na obrazovku jejich součet. Vzhledem k tomu, že výstupem metody *Console.ReadLine* je hodnota typu *string*, musíme před vlastním výpočtem převést řetězce na nějaký reálný datový typ, např. *double*,

```
static void Main(string[] args)
{
    //Nejprve načteme čísla jako řetězce
    Console.Write("Zadejte první cislo: ");
    string cislo1 = Console.ReadLine();
    Console.Write("Zadejte druhé cislo: ");
    string cislo2 = Console.ReadLine();

    //Nyní převedeme řetězce na čísla
    double c1 = Convert.ToDouble(cislo1);
    double c2 = Convert.ToDouble(cislo2);

    //Nakonec čísla sečteme a vypíšeme na obrazovku
    double vysledek = c1 + c2;
    Console.WriteLine("Výsledek: " + vysledek);
}
```

Obrázek 6.5: Načtení čísel

Všimněte si, že pro výpis instrukcí před vlastním voláním metody *Console.ReadLine* používáme metodu *Console.Write*, která na rozdíl od metody *Console.WriteLine* neposune kurzor na nový řádek, tj. vlastní číslo zadáváme na stejném řádku jako samotný popisek.

6.3 Práce se soubory

Dosud měly naše programy tu vlastnost, že si veškerá data uchovávaly pouze v paměti počítače a nebyly schopny si data uložit tak, aby si je mohly znovu načíst při dalším spuštění. Vezměme například program z minulé kapitoly, který evidoval přihlášky studentů. V praxi by takovýto program byl v podstatě nepoužitelný, protože by do něj uživatel zadal přihlášky a nesměl by jej vypnout, protože by jinak o pracně zadaná data přišel. Z toho důvodu je namístě se naučit jeden ze způsobů, jak si program může data uložit, aby si je mohl při dalším spuštění znovu načíst.

Dozvíte se, jak vytvořit program, který zapíše data do textového souboru a zároveň bude schopen data z textového souboru načíst. Textovým souborem myslíme standardní textový

soubor s příponou *.txt*, se kterým můžete pracovat např. prostřednictvím programu Poznámkový blok. Ukládání dat pro další spuštění programu není jediným důvodem, proč je důležité se naučit pracovat s textovými soubory. Představte si, že máte program, který bude např. počítat mzdy brigádníkům, kteří vypomáhali ve skladu. V textovém souboru budete mít strukturovaně zapsané informace o odpracovaných hodinách za jednotlivé dny. Tento soubor mohl vytvořit uživatel sám nebo může být vytvořen např. jiným programem a bude obsahovat stovky řádek dat.

Rozhodně nebudeme chtít, aby uživatel ručně zadával prostřednictvím konzole data, která má již vhodným způsobem předpřipravena v textovém souboru. V některých případech může být dat takové množství, že jejich ruční zadání do programu prostřednictvím konzole je v podstatě nemožné. Budeme tedy chtít uživateli umožnit, aby do programu nahrál data z textového souboru. Kromě nahrání dat do programu mu budeme chtít také umožnit, aby si mohl nechat určitý výstup programu zapsat do textového souboru, aby s daty mohl dále pracovat např. v jiných programech, nebo si je zkrátka mohl nechat uložené v samostatném textovém souboru.

Začněme nejprve ukázkou toho, jak můžeme uložit nějaký řetězec do souboru. Ukážeme si dvě metody pro zápis do souboru. První je metoda *File.WriteAllText*, která do souboru se jménem zadaným v prvním parametru zapíše daný řetězec, který je zadaný jako druhý parametr. Pokud soubor již existuje, tak ho tato metoda nejprve smaže. Druhou metodou je metoda *File.AppendAllText*, která se odlišuje tím, že pokud soubor již existuje, potom daný řetězec přidá k existujícímu souboru. Abychom však mohli metody pro práci se soubory používat, musíme nejprve do našeho programu importovat jmenný prostor, ve kterém se tyto metody nacházejí pomocí příkazu *using Systém.IO* na začátku programu.

```
using System;
using System.IO;

namespace MojeAplikace2
{
    Počet odkazů: 0
    class Program
    {
        Počet odkazů: 0
        static void Main(string[] args)
        {
            string s1 = "Toto řetězec bude uložen do souboru.";
            File.WriteAllText("pokus.txt", s1);

            string s2 = "Tento řetězec bude přidán na konec souboru.";
            File.AppendAllText("pokus.txt", s2);
        }
    }
}
```

Obrázek 6.6: Zápis do souboru

Nyní si ukážeme, jak můžeme textový soubor načíst do proměnné a její obsah pak vypsat na obrazovku. K tomuto účelu budeme využívat *File.ReadAllText*, která obsah celého souboru načte jako řetězec a vrátí ho jako návratovou hodnotu.

```
static void Main(string[] args)
{
    if (File.Exists("pokus.txt"))
    {
        string retezec = File.ReadAllText("pokus.txt");
        Console.WriteLine(retezec);
    }
    else
    {
        Console.WriteLine("Soubor neexistuje");
    }
}
```

Obrázek 6.7: Načtení souboru

Všimněte si v uvedeném příkladu podmínky a zejména metody *File.Exists*, která testuje, zda soubor se zadaným názvem existuje, pokud ano vrací tato metoda logickou hodnotu *true*, jinak vrací *false*. Pokud bychom opomněli otestovat, zda soubor existuje, program by skončil chybou. Vždy je proto z uživatelského hlediska otestovat, zda daný soubor existuje.

Určitým nedostatkem metody *File.ReadAllText* může být to, že nám načte celý obsah souboru do jednoho řetězce. Představme si situaci, kdy v daném textovém souboru máme uložen seznam několika čísel tak, že každé číslo je uvedeno vždy na novém řádku. Našim úkolem by bylo zjistit celkový součet všech čísel v souboru. Je zřejmé, že pro tento účel bude nutné rozdělit obsah souboru na jednotlivé řádku. K takovému účelu se hodí metoda *File.ReadAllLines*, která vrací obsah načteného souboru jako pole řetězců, kdy každý prvek výsledného pole odpovídá příslušnému řádku v souboru. Nyní si ukážeme, jak by takový program mohl vypadat.

```
static void Main(string[] args)
{
    if (File.Exists("pokus.txt"))
    {
        string[] pole = File.ReadAllLines("pokus.txt");
        double suma = 0;
        foreach (string radek in pole)
        {
            double cislo = Convert.ToDouble(radek);
            suma += cislo;
        }
        Console.WriteLine("Součet čísel je: " + suma);
    }
    else
    {
        Console.WriteLine("Soubor neexistuje");
    }
}
```

Obrázek 6.8: Načtení souboru po řádcích

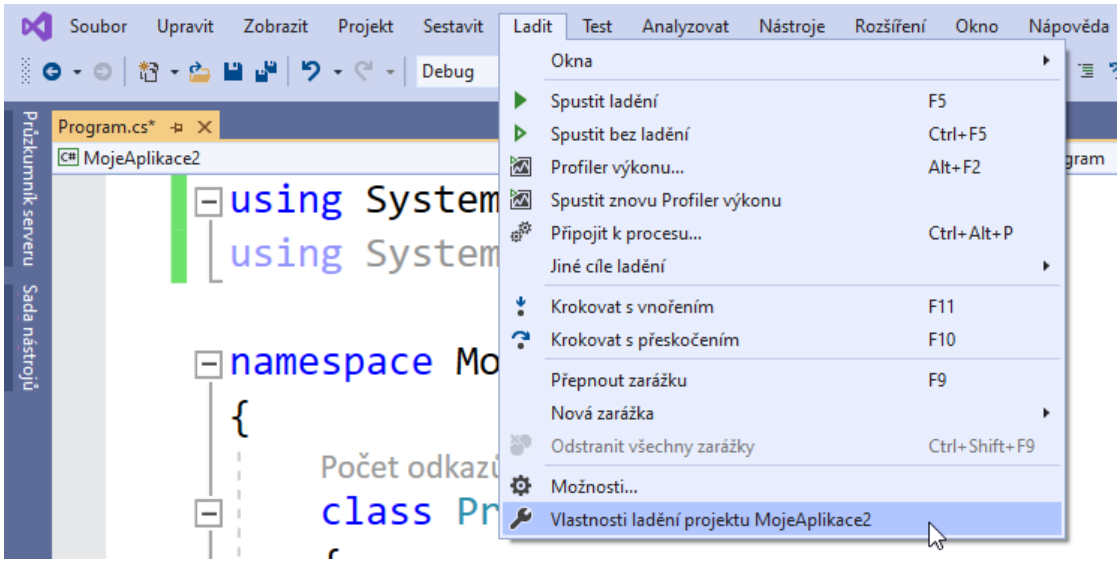
6.4 Předávání parametrů na příkazovém řádku

Prozatím jsme si ukázali dvě možnosti, jak načíst data mimo vlastní kód programu, a to buď z klávesnice nebo ze souboru. Další často používaná možnost, zejména pro konzolové aplikace, je pomocí parametrů na příkazovém řádku. Jistě jste si už všimli, že při vytvoření projektu se automaticky generuje metoda *main*, která má definovaný parametr *args*, který má datový typ pole řetězců. A právě tento parametr se používá pro čtení parametrů zadaných na příkazovém řádku. Ukážeme si nyní, jak zjistit počet zadaných parametrů a jak vypsat na obrazovku jejich hodnoty.

```
static void Main(string[] args)
{
    Console.WriteLine("Počet parametrů: " + args.Length);
    for (int i = 0; i < args.Length; i++)
    {
        Console.WriteLine("Parametr " + i + " = " + args[i]);
    }
}
```

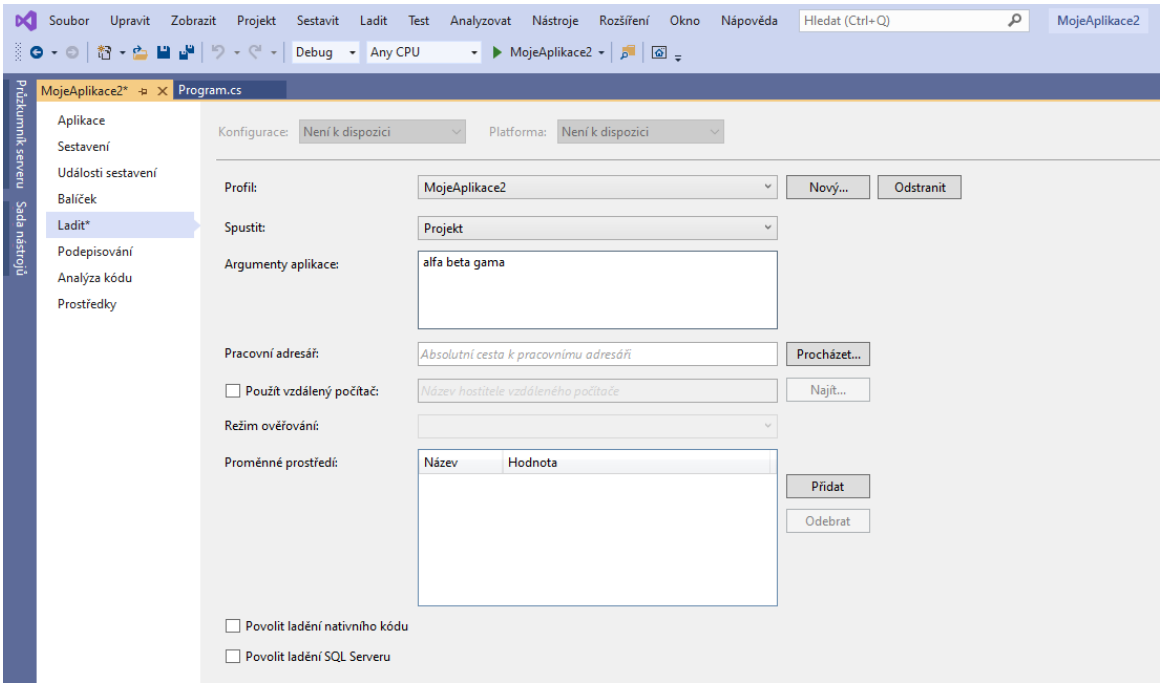
Obrázek 6.9: Parametry příkazového řádku

Když výše uvedený program spustím z prostředí MS Visual Studia, vypíše nám program, že počet parametrů je roven nule. Nyní si ukážeme, jak můžeme v MS Visual Studiu definovat parametry na příkazové řádku. Toho docílíme přes hlavní menu *Ladit* → *Vlastnosti ladění projektu*.



Obrázek 6.10: Vlastnosti ladění projektu

Následně se přepneme vlevo na kartu *Ladit* a v sekci Argumenty aplikace zadáme příslušné parametry oddělené mezerou. V našem případě jsme zadali tři hodnoty „alfa beta gama“.



Obrázek 6.11: Zadávání parametrů na příkazovém řádku

Pokud nyní spustíme naši aplikaci, vypíše nám na obrazovky, že počet parametrů je 3 a zároveň zobrazí jejich hodnoty. Nyní si ještě zkusíme vytvořit aplikaci, která nám spočítá součet čísel zadaných na příkazovém řádku. Pro testování zadáme v MS Visual Studiu do políčka *Argumenty aplikace* „2 3“.

```
static void Main(string[] args)
{
    if (args.Length >= 2)
    {
        double cislo1 = Convert.ToDouble(args[0]);
        double cislo2 = Convert.ToDouble(args[1]);
        double vysledek = cislo1 + cislo2;
        Console.WriteLine(cislo1 + " + " + cislo2 + " = " + vysledek);
    }
}
```

Obrázek 6.12: Součet parametrů příkazového řádku

Všimněte si, že před vlastním spuštěním testujeme, zda jsou na příkazovém řádku zadány alespoň dva parametry. Následně potom tyto parametry převedeme na čísla a ty již pak jen sečteme a vypíšeme na obrazovku.

K ZAPAMATOVÁNÍ

Parametry příkazového řádku se používají nejčastěji pro předání umístění konfiguračního, vstupního či výstupního souboru a pro aktivaci rozšířených funkcí programu.

OTÁZKY

1. Pokud máme dvě proměnné typu *string* obsahující hodnoty „Alfa“ a „Beta“, potom po aplikaci operátoru plus na tyto dvě proměnné, bude výsledkem
    a. Syntaktická chyba
    b. Řetězec „Alfa Beta“
    c. Řetězec „AlfaBeta“
2. Který příkaz vypíše na obrazovku hodnotu celočíselné proměnné *cislo*?
    a. Console.WriteLine(cislo)
    b. Console.WriteLine(“cislo”)
    c. Console.WriteLine(“{1}”, cislo)
3. Který příkaz načte textový vstup od uživatele z klávesnice?
    a. Console.Get()
    b. Consloe.GetLine()
    c. Console.ReadLine()
4. Který příkaz použijeme, pokud aby se kurzor po vypsání textu přesunul na další řádek?
    a. Console.WriteLine()
    b. Consloe.Write()
    c. Console.WriteLineNoEnter()
5. Pokud se pokusíme zapsat text do souboru pomocí metody *File.WriteAllText* a tento soubor již existuje
    a. Existující soubor bude nejprve smazán
    b. Text se připojí na konec souboru
    c. Program skončí chybou
6. Pokud se pokusíme zapsat text do souboru pomocí metody *File.AppendAllText* a tento soubor již existuje
    a. Existující soubor bude nejprve smazán
    b. Text se připojí na konec souboru
    c. Program skončí chybou
7. Pokud se pokusíme přečíst text ze souboru pomocí metody *File.ReadAllText* a tento soubor neexistuje
    a. Program po spuštění oznámí, že soubor neexistuje a zeptá se na název nového souboru
    b. Metoda vrátí prázdný řetězec
    c. Program skončí chybou
8. Metoda *File.Exists* vrací návratovou hodnotu s datovým typem
    a. int
    b. bool
    c. string

9. Jak se nazývá parametr metody *main*, který slouží k předávání parametrů na příkazovém řádku?
    a. args
    b. params
    c. cmdln
10. První parametr příkazového řádku má index
    a. 0
    b. 1
    c. Parametry nejsou indexovány

SHRNUTÍ KAPITOLY

V této kapitole jsme si ukázali základní možnosti, jak může program komunikovat s vnějším světem. Nejprve jsme se zabývali zobrazením výstupu programu na obrazovku, přičemž jsme se zaměřili na možnosti formátování výstupu tak, aby odpovídal potřebám uživatele. Následně jsme si ukázali, jak můžeme načíst vstup od uživatele z klávesnice. Potom jsme přešli na ukázku práce s textovými soubory, a to jak pro načtení dat, tak i pro zápis dat do souboru. Kapitolu jsme ukončili demonstrací možností předávání vstupních dat pomocí parametrů příkazového řádku včetně ukázky nastavení těchto parametrů v MS Visual Studiu pro účely ladění.

ODPOVĚDI

1. c
2. a
3. c
4. b
5. a
6. b
7. c
8. b
9. a
10. a

7 ALGORITMY

RYCHLÝ NÁHLED KAPITOLY

Tato kapitola již nepřináší žádné nové konstrukce jazyka C#, ale je zaměřena spíše prakticky, abychom dokázali aplikovat nabyté znalosti pro vytvoření základních algoritmů pro řešení reálných úloh. Začneme od těch jednodušších, kdy budeme vyhledávat určité prvky v poli, potom se pokusíme setřídit pole prvků, zkusíme si vygenerovat prvočísla. Velká pozornost bude věnována rekurzivním algoritmům, které využívají vlastnosti, že metoda volá sebe sama. Rekurzi budeme demonstrovat na výpočtu součtu řady, faktoriálu, nalezení členů Fibonacciho posloupnosti, největšího společného dělitele dvou čísel a kapitolu zakončíme ukázkou řešení starého hlavolamu Hanojské věže.

CÍLE KAPITOLY

Po prostudování této kapitoly budete umět:

- Samostatně vytvořit jednoduchý algoritmus.
- Zjistit, zda určité číslo existuje v poli.
- Setřídit pole čísel.
- Najít všechna prvočísla do zadané hranice.
- Používat rekurzi.
- Vypočítat součet řady celých čísel pomocí rekurze.
- Vypočítat faktoriál.
- Nalézt členy Fibonacciho posloupnosti.
- Najít největšího společného dělitele dvou čísel.
- Vyřešit hlavolam Hanojské věže.

KLÍČOVÁ SLOVA KAPITOLY

Algoritmus, vyhledávání, třídění, prvočíslo, rekurze, řada, posloupnost, největší společný dělitel, faktoriál, hanojské věže.

7.1 Vyhledávání

První algoritmus, který si ukážeme bude sloužit pro nalezení určitého prvku v poli čísel. Algoritmus budeme implementovat v metodě, která bude vracet index v poli, kde se daný prvek nachází. Pokud pole hledaný prvek nebude obsahovat, metoda vrátí hodnotu -1.

```
static void Main(string[] args)
{
    int[] pole = { 2, 6, 1, 9, 7 };
    int vysledek = Najdi(pole, 9);
    Console.WriteLine("Pozice prvku: " + vysledek);
}

Počet odkazů: 1
static int Najdi(int[] a, int cislo)
{
    int pozicePrvku = -1;
    for (int i = 0; i < a.Length; i++)
    {
        if (a[i] == cislo)
        {
            pozicePrvku = i;
            break;
        }
    }
    return pozicePrvku;
}
```

Obrázek 7.1: Algoritmus vyhledávání

Algoritmus je velice jednoduchý, nejprve nastavíme výchozí návratovou hodnotu -1 do proměnné *pozicePrvku*, následně v cyklu projdeme všechny prvky pole a pokud se některý prvek rovná hledanému číslu zapíšeme aktuální hodnotu iterační proměnné i do proměnné *pozicePrvku* a tu potom na konci cyklu vrátíme jako návratovou hodnotu. Všimněte si zejména příkazu *break*, který je použit v případě nalezení hledaného prvku. Tento příkaz slouží k předčasnému ukončení cyklu, protože při nalezení prvku již nemá smysl hledat dále. Příkaz *break* lze použít u libovolného cyklu, nejen pro cyklus *for*. V této souvislosti je vhodné se zmínit i o příkazu *continue*, který ukončí aktuální iteraci cyklu, avšak v cyklu pokračuje dále.

K ZAPAMATOVÁNÍ

Příkazy *break* a *continue* umožňují předčasně ukončit provádění cyklu.

7.2 Třídění

Existuje celá řada algoritmů pro třídění prvků v poli, které se liší svou složitostí a rychlostí. My si zde ukážeme ten nejjednodušší algoritmus, který sice není tak efektivní, zvláště pro větší počet prvků, ale je velmi jednoduchý pro pochopení. Základní myšlenka tohoto algoritmu spočívá v tom, že budeme hledat vždy minimum z doposud nesetříděné části pole, čímž postupně docílíme toho, že prvky v poli budou seřazeny od nejmenšího po největší.

```
static void Main(string[] args)
{
    int[] pole = { 2, 6, 1, 9, 7 };
    Setrid(pole);
    foreach (int x in pole) Console.WriteLine(x);
}

Počet odkazů: 1
static void Setrid(int[] a)
{
    for (int i = 0; i < a.Length - 1; i++)
    {
        for (int j = i + 1; j < a.Length; j++)
        {
            if (a[j] < a[i])
            {
                int pom = a[i];
                a[i] = a[j];
                a[j] = pom;
            }
        }
    }
}
```

Obrázek 7.2: Třídění

Výše uvedený algoritmus sestává ze dvou vnořených cyklů. První cyklus s iterační proměnnou *i* určuje index prvku, do něhož chceme umístit minimu z dosud nesetříděné části pole. Dosud nesetříděnou část pole reprezentuje druhý vnořený cyklus s iterační proměnnou *j*, který začíná indexem prvku o jeden větší, než je aktuální hodnota iterační proměnné *i*. Uvnitř druhého cyklu zjišťujeme, zda prvek s indexem *j* není menší než doposud nalezené minimum prvku s indexem *i*. Pokud je tato podmínka splněna, zaměníme vzájemně tyto dva prvky. K samotné záměně využíváme pomocnou proměnnou *pom*. Všimněte si ještě, že první cyklus končí indexem předposledního prvku, je tomu tak proto, že pokud jsme setřídili (*n*-1) prvků, tak tím musí být setříděn i ten poslední prvek.

7.3 Eratosthenovo síto

Eratosthenovo síto je jednoduchý algoritmus, který hledá prvočísla v daném intervalu. Algoritmus počítá s tím, že násobky prvočísel již nejsou prvočísla (jsou dělitelná i jiným číslem). Nejdříve objasním prvočísla. Jsou to čísla dělitelná jen jedničkou a sama sebou. Avšak číslo jedna není prvočíslo, proto algoritmus hledá od 2 do *N*. Pokud bychom chtěli najít všechna prvočísla v nějakém intervalu bez Eratosthenova síta, museli bychom ověřovat, jestli dané číslo není dělitelné beze zbytku některým menším číslem, což by bylo velmi náročné. Tento algoritmus si ale škrtá všechny násobky prvočísel, a tak je podstatně rychlejší.

Mějme například řadu čísel do patnácti a víme, že 2 je prvočíslo. Tím pádem všechny jeho násobky jíž prvočísly být nemohou, jelikož jsou už dělitelné dvojkou. Můžeme si je tedy vyškrtnout.

2, 3, ~~4~~, 5, ~~6~~, 7, ~~8~~, 9, ~~10~~, 11, ~~12~~, 13, ~~14~~, 15

Nyní se posuneme na další nevyškrtnuté číslo, čímž je číslo 3 a opět vyškrtneme všechny jeho násobky.

2, **3**, ~~4~~, 5, ~~6~~, 7, ~~8~~, ~~9~~, ~~10~~, 11, ~~12~~, 13, ~~14~~, ~~15~~

Opět se posuneme na další nevyškrtnuté číslo, čímž je číslo 5 a vyškrtneme všechny jeho násobky.

2, **3**, ~~4~~, **5**, ~~6~~, 7, ~~8~~, ~~9~~, ~~10~~, 11, ~~12~~, 13, ~~14~~, ~~15~~

Opět se posuneme na další nevyškrtnuté číslo, čímž je číslo 7 a vyškrtneme všechny jeho násobky.

2, **3**, ~~4~~, **5**, ~~6~~, **7**, ~~8~~, ~~9~~, ~~10~~, 11, ~~12~~, 13, ~~14~~, ~~15~~

Opět se posuneme na další nevyškrtnuté číslo, čímž je číslo 11 a vyškrtneme všechny jeho násobky.

2, **3**, ~~4~~, **5**, ~~6~~, **7**, ~~8~~, ~~9~~, ~~10~~, **11**, ~~12~~, 13, ~~14~~, ~~15~~

Opět se posuneme tentokrát na poslední nevyškrtnuté číslo, čímž je číslo 13 a všechna čísla, která zůstala nevyškrtnutá jsou prvočísla.

2, **3**, 4, **5**, ~~6~~, **7**, ~~8~~, ~~9~~, ~~10~~, **11**, ~~12~~, **13**, ~~14~~, ~~15~~

Tento algoritmus můžeme ještě vylepšit. Je totiž zbytečné procházet čísla až do horní hranice *N*. Můžeme je procházet jen do odmocniny z *N*. Pokud totiž budeme mít číslo zapsané jako *x* * *y*, tak *x* a *y* bude vždy menší nebo rovno než odmocnina z *N*. Nyní můžeme přikročit k zápisu samotného algoritmu.

```
static void Main(string[] args)
{
    int n = 16;
    bool[] cisla = new bool[n];
    cisla[0] = true;
    cisla[1] = true;
    for (int i = 2; i < Math.Sqrt(n); i++)
    {
        if (cisla[i]) continue;
        for (int j = 2 * i; j < n; j += i)
        {
            cisla[j] = true;
        }
    }
    for (int i = 0; i < n; i++)
    {
        if (!cisla[i])
        {
            Console.WriteLine(i);
        }
    }
}
```

Obrázek 7.3: Eratosthenovo síto

K uložení informace o tom, zda bylo číslo vyškrtnuto využíváme pole typu *bool*, kdy logická hodnota *true* znamená, že číslo s tímto indexem bylo vyškrtnuto. Zde je třeba si jen dát pozor na to, že pole začínají indexem 0, proto musíme kapacitu pole nastavit o 1 více než je horní hranice prvočísel, které hledáme. Rovněž si všimněte praktického využití příkazu *continue*, pomocí kterého přeskakujeme čísla, která už byla vyškrtnuta.

7.4 Rekurze

Rekurze je programovací technika, při níž je určitá metoda znovu volána dříve, než je dokončeno její předchozí volání. Použití rekurze může u některých úloh vést ke stručnému a matematicky elegantnímu řešení. Nevede ale nutně k řešení optimálnímu. Použití rekurze

vede obvykle k jinému rozložení využití prostředků přidělených programu operačním systémem, případně k jejich rychlejšímu vyčerpání, proto se při optimalizaci programu většinou snažíme rekurzi omezit nebo odstranit. Nyní si ukážeme příklady několika algoritmů, kde lze využít rekurzi.

K ZAPAMATOVÁNÍ

Každou rekurzi lze nahradit cyklem.

7.4.1 SOUČET ŘADY

Začneme velmi jednoduchou úlohou, kdy bychom chtěli vypočítat součet řady celých čísel od jedné po *n*. Nejprve si ukážeme, jak bychom takovou úlohu řešili bez rekurze pomocí cyklu.

```
static void Main(string[] args)
{
    int vysledek = SoucetRady(5);
    Console.WriteLine(vysledek);
}

Počet odkazů: 1
static int SoucetRady(int n)
{
    int suma = 0;
    for (int i = 1; i <= n; i++)
    {
        suma += i;
    }
    return suma;
}
```

Obrázek 7.4: Součet řady

Nyní si tento kód zkusíme přepsat pomocí rekurze. Využijeme přitom vztahu, že součet řady od 1 do *n* je roven součtu čísla *n* a součtu řady od 1 do (*n*-1). A opět analogickým způsobem rozložíme součet řady od 1 do (*n*-1).

```
static void Main(string[] args)
{
    int vysledek = SoucetRadyRekurze(5);
    Console.WriteLine(vysledek);
}

Počet odkazů: 2
static int SoucetRadyRekurze(int n)
{
    if (n > 0)
    {
        return n + SoucetRadyRekurze(n - 1);
    }
    else
    {
        return 0;
    }
}
```

Obrázek 7.5: Součet řady pomocí rekurze

Všimněte si zejména ukončující podmínky, pokud je totiž parametr n menší nebo roven než nula, již nemá smysl dál provádět rozklad a metoda vrátí jako návratovou nulu. Pokud bychom na ukončující podmínku zapomněli volání sebe sama by nikdy neskončilo, respektive skončilo by chybou nedostatku paměti, protože každé volání metody zabírá určitou část paměti. S využitím ternárního operátoru může algoritmus ještě výrazně zjednodušit.

```
static void Main(string[] args)
{
    int vysledek = SoucetRadyRekurze(5);
    Console.WriteLine(vysledek);
}

Počet odkazů: 2
static int SoucetRadyRekurze(int n)
{
    return n > 0 ? n + SoucetRadyRekurze(n - 1) : 0;
}
```

Obrázek 7.6: Součet řady pomocí rekurze a ternárního operátoru

7.4.2 FAKTORIÁL

Výpočet faktoriálu je analogický s algoritmem výpočtu součtu řady s tím rozdílem, že čísla mezi sebou nesčítáme, ale násobíme. Začneme opět algoritmem bez rekurze využívající cyklus.

```
static void Main(string[] args)
{
    int vysledek = Faktorial(5);
    Console.WriteLine(vysledek);
}

Počet odkazů: 1
static int Faktorial(int n)
{
    int soucin = 1;
    for (int i = 1; i <= n; i++)
    {
        soucin *= i;
    }
    return soucin;
}
```

Obrázek 7.7: Faktoriál

Zde si pouze všimněte, že výchozí hodnota proměnné součin je jedna, protože neutrálním prvkem pro násobení je číslo jedna. Nyní si zkusme přepsat tento algoritmus pomocí rekurze. Využijeme přitom obdobného vztahu, že faktoriál *n* je roven součinu čísla *n* a faktoriálu (*n*-1). A opět analogickým způsobem rozložíme součet řady od 1 do (*n*-1).

```
static void Main(string[] args)
{
    int vysledek = FaktorialRekurze(5);
    Console.WriteLine(vysledek);
}

Počet odkazů: 2
static int FaktorialRekurze(int n)
{
    if (n > 0)
    {
        return n * FaktorialRekurze(n - 1);
    }
    else
    {
        return 1;
    }
}
```

Obrázek 7.8: Faktoriál pomocí rekurze

Opět si všimněte ukončující podmínky pro *n* menší nebo rovno nule, tentokrát je však faktoriál nuly roven 1. Nakonec si zkusíme tuto metodu zjednodušit ještě s využitím ternárního operátoru.

```
static void Main(string[] args)
{
    int vysledek = FaktorialRekurze(5);
    Console.WriteLine(vysledek);
}

Počet odkazů: 2
static int FaktorialRekurze(int n)
{
    return n > 0 ? n * FaktorialRekurze(n - 1) : 1;
}
```

Obrázek 7.9: Faktoriál pomocí rekurze a ternárního operátoru

7.4.3 FIBONACCIHO POSLOUPNOST

Jako Fibonacciho posloupnost je v matematice označována nekonečná posloupnost celých čísel, začínající 0, 1, 1, 2, 3, 5, 8, 13, 21, …, kde první dvě čísla 0 a 1 jsou dána a každé další číslo je součtem dvou předchozích. Fibonacciho posloupnost byla poprvé popsána italským matematikem Leonardem Pisánským (Leonardo z Pisy), známým také jako Fibonacci, k popsání růstu populace králíků za podmínek, že

- První měsíc se narodí jediný pár.
- Nově narozené páry jsou produktivní od druhého měsíce svého života.
- Každý měsíc zplodí každý produktivní pár jeden další pár.
- Králíci nikdy neumírají, nejsou nemocní atd.

V tomto případě začneme algoritmem s využitím rekurze, který přímo odpovídá definici Fibonacciho posloupnosti, tj. každé číslo je součtem dvou předchozích hodnot. Ukončující podmínkou budou první dvě čísla posloupnosti 0 a 1.

```
static void Main(string[] args)
{
    int vysledek = Fibonacci(5);
    Console.WriteLine(vysledek);
}

Počet odkazů: 3
static int Fibonacci(int n)
{
    if (n > 1)
    {
        return Fibonacci(n - 1) + Fibonacci(n - 2);
    }
    else if (n == 1)
    {
        return 1;
    }
    else
    {
        return 0;
    }
}
```

Obrázek 7.10: Fibonacciho posloupnost

Algoritmus můžeme opět zjednodušit pomocí ternárního operátoru, byť zde má podmínka tři větve, ale když se pozorně podíváme na poslední dvě větve, zjistíme metoda vrací 1, pokud je n=1 a pro n=0 vrací nulu, tudíž můžeme sloučit tyto dvě podmínky do jedné a vrátit přímo hodnotu n.

```
static void Main(string[] args)
{
    int vysledek = Fibonacci(5);
    Console.WriteLine(vysledek);
}

Počet odkazů: 3
static int Fibonacci(int n)
{
    return n > 1 ? Fibonacci(n - 1) + Fibonacci(n - 2) : n;
}
```

Obrázek 7.11: Fibonacciho posloupnost s ternárním operátorem

Tento rekurzivní algoritmus nám vždy poskytne správnou hodnotu *n*-tého členu Fibonacciho posloupnosti, ovšem reálně použitelný je pouze pro malá *n*. Už když si zkusíme vypočítat hodnotu Fibonacciho posloupnosti pro *n*=40, můžeme si všimnou jisté časové prodlevy, která se navíc s každým zvýšením hodnoty *n* začne výrazně zvětšovat. Problémem je, že s každým voláním metody Fibonacci se zavolají dvě metody Fibonacci a opět každá z těchto dvou metod zavolá dvakrát sebe sama, jedná se tedy o exponenciální řadu, která roste velmi rychle, čímž je tedy zapříčiněná časová náročnost rekurzivního výpočtu.

Řešením je odstranění rekurze, tedy převedení algoritmu do podoby s využitím cyklu. Toho lze v tomto případě dosáhnout prostřednictvím ukládání již vypočítaných hodnot, které jsou nutné pro další výpočet. V tomto případě zřídíme dvě pomocné proměnné, do kterých budeme ukládat předminulé a minulé vypočítané číslo. Výsledný algoritmus bude vypadat takto.

```
static void Main(string[] args)
{
    long vysledek = FibonacciCyklus(100);
    Console.WriteLine(vysledek);
}

Počet odkazů: 1
static long FibonacciCyklus(int n)
{
    long suma = 0;
    long predchozi = 1;
    long predpredchozi = 0;
    for (int i = 2; i <= n; i++)
    {
        suma = predchozi + predpredchozi;
        predpredchozi = predchozi;
        predchozi = suma;
    }
    if (n == 1) suma = 1;
    return suma;
}
```

Obrázek 7.12: Fibonacciho posloupnost pomocí cyklu

Tento algoritmus je výrazně efektivnější, protože počet kroků nutných pro výpočet roste pouze lineárně s růstem hodnoty *n*. I pro hodnotu *n*=100 nám tento algoritmus poskytne výsledek okamžitě. V případě rekurentního algoritmu bychom pro stejnou hodnotu *n* čekali na výsledek déle, než je odhadováno stáří celého vesmíru. Všimněte si ovšem, že místo datového typu *int* používáme datový typ *long*, je tomu tak proto, že výsledná hodnota Fibonacciho posloupnosti je pro *n*=100 větší než maximální hodnota pro typ *int*.

K ZAPAMATOVÁNÍ

Algoritmus pro nalezení prvků Fibonacciho posloupnosti je typický příklad algoritmu, který je ze své podstaty rekurentní, ale jeho implementace pomocí rekurze výrazně méně efektivní než implementace pomocí cyklu.

7.4.4 EUKLIDŮV ALGORITMUS

Euklidův algoritmus je jeden z nejstarších algoritmů vůbec. Vznikl někdy kolem 300 let př. n. l. a jeho autor není známý (Euklidův se nazývá proto, že se objevuje v jeho knize). Algoritmus umožňuje vypočítat největšího společného dělitele dvou přirozených čísel bez rozkladu na prvočísla. Funguje na principu postupného zmenšování těchto čísel, přičemž však nemění společného dělitele. Když tedy máme čísla *a* a *b*, kde $a > b$ a budeme čísla zmenšovat tak, aby se n. s. dělitel neměnil, nutně dospějeme do fáze, kdy se nám *b* dostane na 0 (protože bylo menší) a to co zbude v *a* je tedy rovnou největší společný dělitel původních čísel. Zkusme si nyní zapsat tento algoritmu s využitím cyklu.

```
static void Main(string[] args)
{
    int vysledek = Euklides(68, 32);
    Console.WriteLine(vysledek);
}

Počet odkazů: 1
static int Euklides(int a, int b)
{
    int c;
    while (b != 0)
    {
        c = b;
        b = a % b;
        a = c;
    }
    return a;
}
```

Obrázek 7.13: Euklidův algoritmus

Nyní si zkusíme tento algoritmus zapsat pomocí rekurze a ternárního operátoru, čímž algoritmus výrazně zjednodušíme.

```
static void Main(string[] args)
{
    int vysledek = EuklidesRekurze(68, 32);
    Console.WriteLine(vysledek);
}

Počet odkazů: 2
static int EuklidesRekurze(int a, int b)
{
    return b == 0 ? a : EuklidesRekurze(b, a % b);
}
```

Obrázek 7.14: Euklidův algoritmus pomocí rekurze

7.4.5 HANOJSKÉ VĚŽE

Hanojské věže (Tower of Hanoi) je matematický hlavolam, který vymyslel francouzský matematik Édouard Lucas v roce 1883. Skládá se ze tří kolíků (věží). Na začátku je na jednom z nich nasazeno několik kotoučů různých poloměrů, seřazených od největšího (vespod) po nejmenší (nahoře). Úkolem řešitele je přemístit všechny kotouče na druhou věž (třetí přitom využije jako pomocnou pro dočasné odkládání) podle následujících pravidel:

- V jednom tahu lze přemístit jen jeden kotouč.
- Jeden tah sestává z vzetí vrchního kotouče z některé věže a jeho položení na vrchol jiné věže.
- Je zakázáno položit větší kotouč na menší.

Vypráví se legenda, že někde ve Vietnamu nebo Indii stojí klášter nebo chrám, v němž jsou hanojské věže se 64 zlatými kotouči. Mniši každý den v poledne za zvuku zvonů slavnostně přemístí jeden kotouč (v jiných verzích probíhá přemisťování nepřetržitě). V okamžiku, kdy bude přemístěn poslední kotouč, nastane konec světa. Vyřešení tohoto hlavolamu pro 64 kotoučů však vyžaduje $2^{64}-1=18\ 446\ 744\ 073\ 709\ 551\ 615$ tahů, takže i kdyby mniši stihli provést jeden tah každou sekundu (a postupovali nejkratším možným způsobem), trvalo by jim vyřešení celého hlavolamu přibližně 600 miliard let, takže brzkého konce světa se bát nemusíme.

Řešení pomocí rekurze vychází z úvahy, že řešení musí obsahovat nejméně jeden tah, v němž je přesunut největší kotouč. Ten však lze přesunout jen tehdy, jsou-li všechny ostatní kotouče nasazeny na třetí věži. Označme počet kotoučů n. Nejprve tedy přesuneme $n–1$ kotoučů (všechny kromě největšího) na odkládací věž, pak přesuneme největší kotouč z počáteční věže na cílovou, a nakonec přesuneme $n–1$ kotoučů z odkládací věže na cílovou.

Přesun *n*–1 kotoučů ovšem můžeme vykonat pomocí rekurzívního volání tohoto algoritmu pro *n*–1 kotoučů, protože největší kotouč nám přitom nevadí (na něj můžeme položit jakýkoli jiný, takže můžeme postupovat, jako by nebyl). Přesný postup je tedy následující:

- Je-li *n*>1, pak rekurzívním voláním této procedury přesuneme *n*–1 kotoučů (tj. všechny kromě největšího) z počáteční věže na odkládací.
- Přesuneme největší kotouč z počáteční věže na cílovou.
- Je-li *n*>1, pak rekurzívním voláním této procedury přesuneme *n*–1 kotoučů z odkládací věže na cílovou.

Nyní si ukážeme samotnou implementaci algoritmu pomocí rekurze.

```
static void Main(string[] args)
{
    HanojskeVeze(4, 1, 3, 2);
}

Počet odkazů: 3
static void HanojskeVeze(int n, int odkud, int kam, int pres)
{
    if (n >= 1)
    {
        HanojskeVeze(n - 1, odkud, pres, kam);
        Console.WriteLine("Přemísti disk z věže " + odkud + " na věž " + kam);
        HanojskeVeze(n - 1, pres, kam, odkud);
    }
}
```

Obrázek 7.15: Hanojské věže

Výstupem programu bude seznam jednotlivých kroků, jak přemístit čtyři disky z věže číslo 1 na věž číslo 3:

```
Přemísti disk z věže 1 na věž 2
Přemísti disk z věže 1 na věž 3
Přemísti disk z věže 2 na věž 3
Přemísti disk z věže 1 na věž 2
Přemísti disk z věže 3 na věž 1
Přemísti disk z věže 3 na věž 2
Přemísti disk z věže 1 na věž 2
Přemísti disk z věže 1 na věž 3
Přemísti disk z věže 2 na věž 3
Přemísti disk z věže 2 na věž 1
Přemísti disk z věže 3 na věž 1
Přemísti disk z věže 2 na věž 3
Přemísti disk z věže 1 na věž 2
Přemísti disk z věže 1 na věž 3
Přemísti disk z věže 2 na věž 3
```

PRO ZÁJEMCE

Sami si můžete ověřit, zda je tento postup správný.

OTÁZKY

1. Eratosthenovo síto je algoritmus pro nalezení
   a. Lichých čísel
   b. Sudých čísel
   c. Prvočísel
2. Euklidův algoritmus je algoritmus pro nalezení
   a. Největšího společného dělitele dvou čísel
   b. Prvočísel
   c. Nejmenšího společného násobku dvou čísel
3. Fibonacciho posloupnost je definována jako
   a. Každý člen je roven součtu dvou předchozích členů posloupnosti
   b. Každý člen je roven dvojnásobku předchozího členu posloupnosti
   c. Každý člen je roven přirozenému logaritmu předchozího členu posloupnosti
4. Která realizace algoritmu pro nalezení členů Fibonacciho posloupnosti je efektivnější?
   a. Jsou přibližně stejně efektivní
   b. Pomocí cyklu
   c. Pomocí rekurze
5. K čemu slouží příkaz *continue* v cyklu?
   a. Předčasně ukončí celý cyklus
   b. Předčasně ukončí aktuální iteraci cyklu, ale samotný cyklus dále pokračuje
   c. Přeskočí se kód, který je před příkazem *continue*
6. Co je to rekurze?
   a. Neexistuje
   b. Jiný výraz pro cyklus
   c. Druh algoritmu, kdy v průběhu výpočtu metoda volá sebe sama
7. Lze rekurzi nahradit pomocí cyklu?
   a. Ano
   b. Ne
   c. Ano, ale jen pokud je počet volání rekurentního algoritmu menší než deset
8. Celkový počet iterací vnořených cyklů je roven
   a. Součtu iterací jednotlivých cyklů
   b. Součinu iterací jednotlivých cyklů

   c. Součtu přirozeného logaritmu iterací jednotlivých cyklů
9. Jaký návratový typ může mít metoda počítající faktoriál 20 tak, aby výsledek byl správný?
   a. short
   b. int
   c. long
10. Jak dlouho bude přibližně trvat vyřešení původní verze hlavolamu Hanojské věže pro 64 disků na běžném osobním počítači?
   a. Jeden den
   b. Jeden rok
   c. Běžný počítač tento hlavolam nevyřeší ani za celý lidský život

SHRNUTÍ KAPITOLY

V této kapitole jsme se zaměřili na praktickou aplikaci nabytých znalostí z předchozí kapitol. Začali jsme ukázkou jednoduchého algoritmu pro nalezení indexu prvku v poli, potom jsme si ukázali použití vnořených cyklů pro setřídění pole čísel. Rovněž jsme si ukázali, jak vypsat všechna prvočísla do zadané horní meze. Následně jsme se seznámili s takzvanou rekurzí, kdy metoda při výpočtu volá opakovaně sebe sama. Použití rekurze jsme si demonstrovali na výpočtu součtu řady celých čísel, výpočtu faktoriálu, vypsání členů Fibonacciho posloupnosti, nalezení největšího společného dělitele dvou čísel a kapitolu jsme uzavřeli rekurzivním algoritmem pro řešení známého hlavolamu Hanojské věže.

ODPOVĚDI

1. c
2. a
3. a
4. b
5. b
6. c
7. a
8. b
9. c
10. c

LITERATURA

BORY, P., 2016. *C# – Bez předchozích znalostí*. Brno: Computer Press. ISBN 978-80-251-4686-6.

KNUTH, D., 2014. *Umění programování – 1. díl*. Brno: Computer Press. ISBN 978-80-251-20-25-5.

FORD, S., 2009. *266 tipů a triků pro Microsoft Visual Studio*. Brno: Computer Press. ISBN 978-80-251-2554-0.

NAGEL, C., 2007. *C# 2005 Programujeme profesionálně*. Brno: Computer Press. ISBN 80-251-1181-4.

PARTSKHALADZE, G., 2005. *Technologie programování*. Karviná: SU OPF. ISBN 9788072483167.

VIRIUS, M., 2002. *C# pro zelenáče*. Praha: Neokortex. ISBN 8086330117.

WIRTH, N., 1990. *Algoritmy a štruktúry údajov*. Bratislava, ALFA. ISBN 80-05-00153-3.

WRÓBLEWSKI, P., 2015. *Algoritmy*. Brno: Computer Press. ISBN 978-80-251-4126-7.

SHRNUTÍ STUDIJNÍ OPORY

Tato studijní opora se vás pokusila seznámit se základy algoritmizace a základy programovacího jazyka C# ve vývojovém prostředí Microsoft Visual Studio. Po jejím přečtení byste měli být schopni samostatně implementovat jednoduchý i středně složitý algoritmus v jazyce C# s využitím pravidel strukturovaného programování.

Pokud vás programování zaujalo, mohu vám doporučit studium dalších knih uvedených v Literatuře. Pro dokonalé zvládnutí však nebude stačit studium jakéhokoliv množství literatury. Je třeba zejména vše zkoušet implementovat v příslušném programovacím jazyce.

Po úspěšném zvládnutí základů strukturovaného programování je vhodné se poté poohlédnout po objektovém programování, které je v současné době standardem v programování.

PŘEHLED DOSTUPNÝCH IKON


Čas potřebný ke studiu


Cíle kapitoly


Klíčová slova


Nezapomeňte na odpočinek


Průvodce studiem


Průvodce textem


Rychlý náhled


Shrnutí


Tutoriály

Definice


K zapamatování


Případová studie


Řešená úloha

Věta


Kontrolní otázka


Korespondenční úkol


Odpovědi


Otázky


Samostatný úkol


Další zdroje


Pro zájemce


Úkol k zamyšlení

Název: **Úvod do programování**

Autor: **Ing. Radomír Perzina, Ph.D.**

Vydavatel: Slezská univerzita v Opavě

Obchodně podnikatelská fakulta v Karviné

Určeno: studentům SU OPF Karviná

Počet stran: 114

Tato publikace neprošla jazykovou úpravou.