Inovace úloh v kurzu jazyka C++

Tomáš Bartošík

Bakalářská práce
2019/2020

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky

Univerzita Tomáše Bati ve Zlíně

Fakulta aplikované informatiky

Ústav informatiky a umělé inteligence

Akademický rok: 2019/2020

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	**Tomáš Bartošík**
Osobní číslo:	**A17115**
Studijní program:	**B3902 Inženýrská informatika**
Studijní obor:	**Softwarové inženýrství**
Forma studia:	**Prezenční**
Téma práce:	**Inovace úloh v kurzu jazyka C++**
Téma práce anglicky:	**The Innovation of Assignments in the C++ Programming Course**

Zásady pro vypracování

1. Prostudujte možnosti realizace automatické kontroly správnosti kódu v systému Gitlab.
2. Navrhněte sadu úkolů pro kurz Programování v jazyce C++, které budou zaměřeny na procvičení základů syntaxe, objektového programování, využití knihoven STL a SQLite.
3. Pro jednotlivé úkoly implementujte vzorová řešení, jednotkové testy a testovací scénáře.
4. V systému Gitlab realizujte skripty pro automatickou kontrolu správnosti odevzdaných řešení.
5. Výslednou sadu vyzkoušejte v reálné výuce.

Rozsah bakalářské práce:
Rozsah příloh:
Forma zpracování bakalářské práce: **tištěná/elektronická**

Seznam doporučené literatury:

1. SUTTER, Herb a Andrei ALEXANDRESCU. C++: 101 programovacích technik. Brno: Zoner Press, 2005. Encyklopedie Zoner Press. ISBN 8086815285.
2. MATOUŠEK, David. C++: výukový kurz. Brno: Computer Press, 2018. ISBN 9788025149065.
3. LANGR, Jeff a Michael SWAINE. Modern C++ programming with test-driven development: code better, sleep better. Dallas, Texas: The Pragmatic Bookshelf, [2013]. ISBN 1937785483.
4. MYERS, Glenford J., Tom BADGETT, Todd M. THOMAS a Corey SANDLER. The art of software testing. 2nd ed. Hoboken, N.J.: John Wiley, c2004. ISBN 0-471-46912-2.
5. HALDAR, Sibsankar. Inside SQLite. Sebastopol, Calif.: O'Reilly, 2007. ISBN 9780596550066.

Vedoucí bakalářské práce: **Ing. Tomáš Dulík, Ph.D.**
Ústav informatiky a umělé inteligence

Datum zadání bakalářské práce: **28. listopadu 2019**
Termín odevzdání bakalářské práce: **15. května 2020**

doc. Mgr. Milan Adámek, Ph.D.
děkan

prof. Mgr. Roman Jašek, Ph.D.
ředitel ústavu

Ve Zlíně dne 9. prosince 2019

Prohlašuji, že

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

Prohlašuji,

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

Ve Zlíně, dne
23. července 2020

Tomáš Bartošík, v. r.
podpis diplomanta

ABSTRAKT

Cílem bakalářské práce a související studie je rekonstrukce existujících a následná tvorba nových úkolů pro náplň cvičení programovacího jazyka C++, jejichž kontrola je založena na principu jednotkových testů v rámci síťového repozitáře včetně následného rozboru paměti. Jednotlivé úlohy jsou zaměřeny na klíčové součásti doporučených programovacích praktik a syntaxe při vývoji aplikací.

Kromě samotné syntaxe jazyka C++, základů objektového programování a polymorfismu je student seznámen s využitím knihovny STL (Standard Template Library) a aplikuje jednoduché příkazy jazyka SQL při práci s knihovnou SQLite. Studentovi je poskytnuta základní struktura každého úkolu, pomocí které implementuje předpokládané chování aplikace. Jednotlivé úlohy jsou obohaceny o návod v českém a anglickém jazyce a dílčí prvky základní struktury obsahují dokumentaci ve formátu generátoru Doxygen.

Klíčovou součástí výzkumu je rovněž použití zpracovaných úloh v reálné výuce za účelem vytvoření statistiky úspěšnosti studentů při zpracování vybraných úkolů a získání námětů pro následné vylepšení poskytnutých učebních podkladů. Studie je dále rozšířena o výzkum popisující kontrolní metody při rozboru kódu na vybraných webových stránkách s online kurzy programování v jazyce C++.

Klíčová slova: programování, testování počítačů, automatizace, výukový software

ABSTRACT

The purpose of this bachelor's thesis and related studies involves reconstruction of existing and additional creation of new assignments for C++ programming lessons that are meant to be unit tested via network repository and include further memory analysis. Assignments are focused on key components of recommended programming practices and syntax used for application development.

Aside of C++ language syntax, basics of object oriented programming and polymorphism, student is acquainted with the usage of STL (Standard Template Library) and applies simple

commands of SQL language when working with SQLite library. Student has the base structure of each assignment at disposal, which is used for implementation of required application behavior. Every assignment includes a manual in czech and english language and partial segments of the base structure include documentation in Doxygen generator format.

Next crucial part of this research involves using the provided programming tasks in practice in order to measure the success rate of students and to obtain an inspiration for further improvements. The thesis is extended by an investigation in control methods of code analysis on selected web sites with online courses of C++ programming.

Keywords: programming, computers testing, automation, educational software

Poděkování

Mé díky patří panu Ing. Tomáši Dulíkovi, Ph.D., který mi poskytl odborné vedení při inovaci úloh a zpracování mé bakalářské práce. Děkuji také panu Ing. Janu Dolinay, Ph.D. za prostor k využití vybrané inovované úlohy v rámci kurzu „Programování v jazyce C/C++“ během probíhajícího semestru.

OBSAH

ÚVOD

Předmětem práce je tvorba úloh pro cvičení základů programování v jazyce C++ se zaměřením na výuku správné syntaxe programovacího jazyka a na principy objektově orientovaného programování. Výsledkem je upravená sada úloh, jejichž cílem je seznámit studenta se syntaktickými a sémantickými rozdíly jazyka C++ oproti jazyku C. Student se naučí nejen vytvářet a dále pracovat s objekty a třídami, ale také uplatní využití přetížení operátorů či dědičnosti. Za tím účelem jsem navrhl zcela nové úlohy, které rozšiřují náplň kurzu například o aplikaci šablon tříd a knihovny STL (Standard Template Library) pro práci s funkčními objekty či kontejnery. Dalším tematickým doplňkem je využití knihovny SQLite pro realizaci práce s místní databází nebo algoritmická úloha hledání cesty pro navštívení všech polí volitelně velké šachovnice pomocí šachové figurky jezdce.

Navržené úlohy jsem implementoval ve dvou verzích: v první verzi řešené úkoly zahrnují vzorovou implementaci, v druhé verzi pouze kostru s vynecháním podstatných částí kódu, jehož vypracování je zadáno studentům ve formě samostatné práce. Úlohy obsahují návod v českém a anglickém jazyce, ve zdrojovém kódu je dokumentace vložena v syntaxi generátoru Doxygen. Všechny úlohy, bez závislosti na tom, jedná-li se o verzi s vzorovým řešením či pouze se zadáním, dodržují konzistentní strukturu zápisu zdrojového kódu pro sjednocení stylu a zajištění vyšší přehlednosti.

Oproti předchozí verzi úloh je nově implementována kontrola správné funkce odevzdávaných úkolů, která je založena na jednotkových testech. Jednotkové testy jsou svým pokrytím specifické pro každý z úkolů. Tyto testy, utvářené dle definovaných testovacích scénářů, jsou součástí úloh již při prvním stažení zadání ze síťového repozitáře. Cílem studenta je vypracovat úlohy dle poskytnutého zadání, funkcionalita odevzdaných prací a správnost kódu je posléze ověřována pomocí automatizovaného kontrolního systému GitLab, pro který jsem navrhl testovací skripty. Z tohoto důvodu bylo nezbytné zajistit podporu vypracování a odevzdání úloh jak pro systém Windows, tak pro Linux. Na straně systému GitLab je rovněž prováděna kontrola pomocí nástroje Valgrind, který umožňuje zjištění úniku paměti při jejím nesprávném či chybějícím uvolnění.

Jedním z požadavků zadání je použití nově vytvořených úloh v praktické výuce a shromáždění dat pro tvorbu statistiky úspěšnosti studentů a spokojenosti studentů s testovacími metodami. Tento výzkum je rovněž důležitý z hlediska námětů k následnému vylepšení úloh či testů pro pokrytí všech testovacích případů. Studenti dostali příležitost vyzkoušet vybranou

novou úlohu – byli seznámeni se způsobem získání zadání, s ověřováním jejich pokroku prostřednictvím jednotkových testů a se zobrazením výsledků automatické kontroly po jejich odevzdání.

Pro porovnání nově zavedeného formátu úloh a metod jejich ověřování jsem nakonec provedl také průzkum spojený s principem kontrolních technik online učebních kurzů (nejen) jazyka C++.

I. TEORETICKÁ ČÁST

1 PŘEDPOKLADY PRO VÝBĚR OBSAHU ÚLOH PROGRAMOVACÍHO JAZYKA C++

Základním předmětem inovace existujících úloh a tvorby nových zadání pro kurz výuky programování v jazyce C++ je především vhodný výběr tematických okruhů. Předpokladem úspěšnosti studentů je správné pořadí vybraných témat a přizpůsobená složitost. Obsah lze kategorizovat na osvojení syntaktických základů programování v C++, uplatnění principů objektově orientovaného programování (OOP), využití virtuálních metod a polymorfismu, nebo na práci s knihovnami jako Standard Template Library (STL) nebo SQLite.

1.1 Přístup k základům struktury a syntaxe jazyka C++

Mnoho syntaktických principů, jako například postup deklarace různých používaných proměnných, odpovídá struktuře deklarací v programovacím jazyce C. Proměnné mohou být tvořeny různými datovými typy, které se odlišují svou velikostí v paměti a použitím *[1, str. 138].* Faktor velikosti datového typu je ovlivněn systémem a jeho překladačem, proto například u datového typu *char* předpokládáme minimální velikost 1 Byte. Velikosti jednotlivých datových typů lze snadno zjistit implementací funkce *sizeof()* nebo využitím knihovny *<limits>*.

Samotná deklarace proměnné slouží k jejímu následnému adresování, jedná se o vymezení pracovního prostoru v paměti, aby sem bylo možné ukládat data. V průběhu deklarace proměnné je rovněž určen její název, pomocí kterého lze k danému místu v paměti snadno přistoupit namísto předávání číselné hodnoty adresy *[2, str. 47]*. Základem pojmenovávání v programovacím jazyce C++ je nutnost rozlišení mezi malými a velkými písmeny (case sensitivity), proto tedy názvy „*hodnota*“ a „*Hodnota*“ odkazují na dvě různé proměnné. Názvy mohou obsahovat i číslice a další povolené znaky, jako například symbol podtržítka, číslicí však název začínat nesmí. Současně platí, že název proměnné či funkce nemůže odpovídat názvu vyhrazených klíčových slov (ke kterým patří např. *new*) a nesmí být oddělen mezerou *[3]*.

Definice, na rozdíl od deklarace, rozšiřuje implementaci o tělo funkce, resp. o přiřazení hodnoty. Proměnnou lze přímo definovat bez nutnosti předchozí deklarace, protože definováním je proměnná či funkce také deklarována.

1.1.1 Dělení programu do jednotlivých souborů

Pro zachování konzistence a přehlednosti je vhodné programy členit do hlavičkových a zdrojových (implementačních) souborů, které jsou u programovacího jazyka C++ nejčastěji specifikovány koncovkami *.hpp* (nebo *.h*) a *.cpp*. Hlavičkové soubory obecně slouží pro deklaraci funkcí či šablon a lze je pomocí direktivy *include* zahrnout do zdrojových souborů, kde je obvykle zpracována definice funkcí. Lokálně vytvářené soubory jsou vloženy po uzavření do uvozovek *[1, str. 424]*.

Direktivou *include* lze zahrnout i soubory standardních knihoven, jako je například knihovna *iostream* pro zpracování standardních vstupů či výstupů. Názvy standardních knihoven jsou při vkládání ohraničeny ostrými závorkami. Pomocí této knihovny lze mimo jiné využít operátoru dvou ostrých závorek vlevo (*„<<“)* a standardního výstupního proudu *cout* pro zpracování výstupu *[1, str. 91]*.

1.1.2 Pojednání o jmenných prostorech

Při sjednocování většího množství zdrojových souborů mnohdy dochází ke kolizím způsobeným shodným pojmenováním používaných součástí, kupříkladu vyskytují-li se ve dvou souborech stejnojmenné globální proměnné. Jmenné prostory (namespaces), strukturou připomínající třídu, zabraňují konfliktům mezi shodnými identifikátory. Jmenný prostor lze deklarovat užitím klíčového slova *namespace* a názvem prostoru (identifikátorem), který je však nepovinnou součástí – v případě absence pojmenování vzniká prostor anonymní. Jmenné prostory lze vnořovat, tj. je možné zahrnout jmenný prostor do již existujícího jmenného prostoru, aniž by byl ovlivněn obor viditelnosti identifikátorů v programu. Při využívání identifikátoru mimo jmenný prostor jeho vzniku je třeba jej kvalifikovat, proto se při adresování využívá operátoru dvou dvojteček (*::*) a udání jména prostoru požadované proměnné, případně lze využít direktivy *using* s názvem jmenného prostoru *[4, str. 222-223]*.

1.2 Aplikování principů objektově orientovaného programování

Objektově orientované programování (neboli OOP) vyjadřuje metodiku programování pro tvorbu abstraktního popisu reálně existujících prvků – spočívá tedy ve zjednodušení, resp. zobecnění určitého celku, který může být reprezentován objektem. Každý vytvořený objekt je význačný svými vlastnostmi a vztahy vůči ostatním objektům, základní formu těchto

vztahů odráží porovnání *objekt je*, které představuje podobnost objektu s jiným objektem, a *objekt má*, což reprezentuje vztah obsažení objektu v jiném. Na základě těchto společných vlastností a závislostí lze aplikovat techniky dědičnosti a skládání *[2, str. 216]*.

Na rozdíl od procedurálního programování, kde jsou jednotlivé činnosti programu popsány datovými strukturami a funkcemi, které zajišťují jejich obsluhu, objektově orientované programování je založeno na tvorbě objektů. Přístup procedurálního programování vede ke špatné udržitelnosti v případě komplexních programů, protože programová data a kód, který zajišťuje jejich obsluhu, jsou odděleny. K nedostatkům procedurálního programování patří rovněž mnohdy problematické řešení změn v programu a následně vyšší šance na vytvoření nových chyb. Objekty ovšem zahrnují jak data (atributy), tak obslužné procedury (členské funkce) a jsou proto soběstačné. Dále umožňují uplatnění technik zapouzdřování a skrývání pomocí definice oboru viditelnosti *[5, str. 711-712]*.

1.2.1 Třída a její vlastnosti

Třída v programovacím jazyce C++ představuje uživatelsky definovaný vzor obsahující členské vlastnosti (atributy) a členské funkce, které mohou definovat například proces vytváření, přesunu nebo odstranění. Třídu si lze představit jako jmenný prostor, který obsahuje své členské vlastnosti a funkce. Třída má, z hlediska oboru viditelnosti, v základní konfiguraci nastaveny členské vlastnosti na privátní, to znamená nepřístupné mimo jmenný prostor třídy. Součástí třídy je konstruktor a destruktor, což jsou speciální členské funkce, které zajišťují inicializaci a odstraňování vytvářených součástí. Tvorba třídy je možná pomocí klíčového slova *class*, její název by měl začínat velkým písmenem *[1, str. 450-451]*.

Rozdíl mezi třídou a strukturou v C++ spočívá ve výchozím nastavení oboru viditelnosti. Oproti třídě má struktura (struct) své členy ve výchozí konfiguraci veřejné, může také implementovat členské atributy i členské funkce. Obecně je lze použít pro sloučení členských atributů různých datových typů do jednoho celku. Pro jejich tvorbu se využívá klíčového slova *struct*, po vytvoření instance přistupujeme ke členským proměnným pomocí operátoru tečky (.), v případě tvorby ukazatele šipkou (->). Struktury i třídy lze v C++ využít takřka identicky (kromě rozdílu základní viditelnosti členů), u obou typů lze stejnoměrně využít všech možností programovacího jazyka včetně tvorby konstruktorů a destruktorů, členských funkcí, uplatnění polymorfismu, tvorby šablon nebo například přetěžování operátorů *[2, str. 217-227].*

Konstruktor je speciální členská funkce třídy či struktury, která je automaticky zavolána v případě tvorby objektu třídy, to znamená při zápisu dat objektu do prostoru paměti nebo při vytvoření instance. Slouží pro prvotní inicializaci atributů objektu (nastavení požadovaných hodnot) před tím, než jsou členské atributy dostupné v jiných částech kódu. Název konstruktoru je shodný s názvem třídy. Dále jsou konstruktory specifické tím, že nemají návratovou hodnotu. Obdobně jako jiné členské funkce však mohou přijímat vstupní argumenty, mohou být nakonfigurovány jako vložené (inline) a lze je přetížit. V případě absence vlastní definice konstruktoru je použit výchozí „prázdný“ konstruktor, který nemá žádnou uživatelsky určenou funkcionalitu *[5, str. 746-750]*.

Podobně jako u konstruktoru je destruktor speciální funkce, která je automaticky zavolána, v tomto případě při odstranění vytvořeného objektu. Rovněž je jeho název tvořen názvem třídy nebo struktury, ve které se destruktor nachází, před názvem je však obsažen znak vlnovky (~). Slouží pro vykonání nezbytných procedur při odstraňování objektu, nejčastěji ve formě odstranění dynamicky alokované paměti. Destruktory nemají návratovou hodnotu, nemohou ani přijímat vstupní argumenty *[5, str. 758-760]*.

1.2.2 Obor viditelnosti a spřátelené funkce

K přínosům definování viditelnosti u členů tříd či struktur patří řízení jejich přístupnosti mimo prostor jejich vzniku. Řízení přístupu umožňuje definovat veřejné (public), chráněné (protected) nebo soukromé (private) členy. K veřejně vytvořeným členům, které jsou označeny klíčovým slovem *public*, lze přistoupit kteroukoliv funkcí. Chráněné členy, význačné klíčovým slovem *protected*, lze volat pouze členskými a spřátelenými funkcemi téže třídy, rovněž členskými a spřátelenými funkcemi odvozených tříd, pokud existují. U soukromých členů, které definujeme klíčovým slovem *private*, lze umožnit přístup pouze členským a spřáteleným funkcím stejné třídy *[1, str. 600]*.

U tříd jsou v základní konfiguraci členy nastaveny na soukromé, tj. přístupné pouze v oblasti jejich definice. Toto nastavení slouží k bezpečnému uchování neměnných (invariantních) hodnot při případných budoucích změnách v programu. V případě veřejné konfigurace viditelnosti členů hrozí nežádoucí, mnohdy i nepředvídatelné změny stavu součástí třídy. U chráněných členů může být konzistence abstraktního stavu narušena tvorbou odvozené třídy, která může neměnné hodnoty členů rodičovské třídy upravit a v případě chybného scénáře tuto konzistenci poškodit *[6, str. 87]*.

Funkce, která je označena klíčovým slovem *friend*, je tzv. spřátelená funkce, která má přístup ke členům třídy nebo struktury, ale samotná není její součástí. Jsou-li v programu definovány dvě nezávislé komponenty obsahující privátní členy, lze členskou funkci jedné z komponent v druhé označit za spřátelenou a tím jí umožnit přístup ke svým členským atributům, které v jiném případě nejsou mimo mateřskou komponentu přístupné. Tuto strategii lze využít například při přetěžování operátorů *[1, str. 571-572]*.

1.2.3 Statické členy

Statické členy, jmenovitě statické členské atributy a statické členské funkce, umožňují tvorbu své definice ve smyslu zobrazení třídy či struktury namísto vytváření instance. Při přiřazování hodnoty statickému členskému atributu není hodnota ukládána ve formě instance třídy, samotná instance nemusí ani existovat – proto je možné vytvořit členský atribut nebo členskou funkci, která nepatří žádné instanci. Statické členské funkce nevykonávají operace nad vytvářenými instancemi třídy, nýbrž pouze nad členskými statickými atributy.

Statický členský atribut nebo členskou funkci třídy je možné vytvořit pomocí klíčového slova *static*. U statických členských atributů platí, že v paměti je uchována pouze jediná jejich kopie bez ohledu na počet instancí třídy, které mohou existovat – všechny instance pak tento statický atribut sdílí. Definici hodnoty statického členského atributu je třeba určit mimo deklarační blok třídy, všem neinicializovaným statickým členským atributům je automaticky přiřazena nulová hodnota. Životnost statického členského atributu odpovídá celé životnosti běhu programu a tyto atributy vznikají již před případnou tvorbou instancí dané třídy či struktury. Z této skutečnosti plyne výhoda statických členských funkcí – mohou přistoupit ke statickým členským atributům ještě před samotnou tvorbou instance třídy. Volání statické členské funkce lze zajistit odlišnou notací oproti nestatickým členským funkcím, a sice přímým uvedením názvu třídy nebo struktury, připojením operátoru dvou dvojteček (scope resolution operator) a uvedením názvu statické funkce *[5, str. 812-818]*.

1.2.4 Tvorba instancí

Pokud lze uvažovat o třídě jako o šabloně nebo vzoru tvořící objekt a jeho podobu, pak instance třídy představuje tvorbu objektu z této šablony. Všechny instance určité třídy nebo struktury sdílí členské vlastnosti dle vzoru vytvořené třídy, rovněž umožňují vykonání stejných operací dle existujících členských funkcí. Při vytváření instance je třeba uvést název třídy, která reprezentuje nově vytvářený objekt a pojmenování (identifikace) nově vytvářené

instance. Je-li přetížen výchozí konstruktor třídy a obsahuje-li parametry (například výchozí hodnoty členských atributů), jsou tyto hodnoty uvedeny v kulatých závorkách, které následují po pojmenování instance. Tento způsob tvorby vyhrazuje potřebné místo v paměti (na zásobníku) *[7]*.

Při vytváření instancí lze rovněž uplatnit dynamické alokování paměti prostřednictvím klíčového slova *new*. Tato metoda vyhradí potřebné místo v paměti (na haldě), které zůstává rezervované po celou dobu běhu programu. Při vytváření instance třídy pomocí *new* je správné alokovanou část v paměti opět uvolnit, jinak dochází k úniku paměti. Toho lze docílit využitím klíčového slova *delete*. V případě obdoby této metody při vytváření nebo mazání celého pole nových instancí lze ke klíčovým slovům připojit hranaté závorky, tj. zavolat *new[]* a *delete[]*. Pokud je přetížen *new* pro vlastní definovaný typ, je třeba stejným způsobem přetížit i *delete*, jinak hrozí výskyt úniku paměti. *[6, str. 95-96]*.

1.2.5 Šablony tříd

Šablony lze použít pro tvorbu generických tříd a abstraktních datových typů, rovněž umožňují vytvořit obecnou verzi třídy bez nutnosti využití redundantního kódu pro více typových formátů dat. Kupříkladu namísto nutnosti vytváření třídy obsahující členské funkce pro obsluhu sčítání čísel datového typu *int* a *float* (a přitom muset vytvořit dvě stejné třídy, rozdílné pouze typem vstupního argumentu), je možné využít klíčového slova *template* pro vytvoření šablony. Do ostrých závorek, které vymezují typ pro šablonu, se v případě vytváření šablony tříd uvádí klíčové slovo *class* s identifikačním názvem typu (běžně *T*). Poté se provádí deklarace třídy pro podporu různých datových typů, namísto explicitního uvedení typu (například tedy *int* nebo *float*) se uvádí právě identifikační typ *T [5, str. 996-997]*.

Třída vytvořená jako šablona je zcela regulérní třídou, tudíž využití šablony není příčinou změny běhových mechanismů, které se běžně využívají u tříd. Využití šablony může mít za následek snížení množství vytvářeného kódu, protože kód pro určitou členskou funkci šablony tříd je generován pouze v případě, že je tento člen využit. Členy šablon tříd mohou být datové prvky (ve formě proměnných a konstant), členské funkce, statické členy, členské typy (třídy) či členské šablony (jako například členské šablony tříd). I třídy vytvořené pomocí šablon mohou deklarovat přátelské funkce *[1, str. 669-675]*.

1.3 Uplatnění polymorfismu a virtuálních metod

Polymorfismus neboli mnohotvárnost umožňuje využití jednotného rozhraní pro práci s různými typy objektů. Podstata využití vychází ze skutečnosti, že ukazatel na instanci odvozené třídy je typově kompatibilní s ukazatelem na základní třídu této instance *[8]*.

Typ, který obsahuje virtuální funkce, je označován jako *polymorfní typ*, v jazyce C++ je pro využití polymorfních technik potřeba, aby byly členské funkce označeny klíčovým slovem *virtual*. Se samotnými objekty je vyžadována manipulace prostřednictvím ukazatelů či referencí – je-li přístup k objektům realizován napřímo, pak je přesný typ objektu znám překladači a průběžný (*run-time*) polymorfismus není nutné uplatňovat. Členská funkce odvozené třídy se shodným názvem a sadou stejných typů argumentů, jako u třídy rodičovské, předefinuje (*override*) verzi virtuální funkce, která je obsažena v rodičovské třídě. Pokud není verze volané virtuální funkce explicitně určena, pak je konkrétní funkce pro předefinování vybrána dle vhodnosti k danému objektu, nad kterým k volání dochází. Pokud využíváme mechanismu volání virtuálních funkcí, pak nezáleží na tom, která základní třída (rozhraní) přistupuje k objektu – vždy je volána stejná funkce *[1, str. 587]*.

U instancí tříd, kde je uplatňován polymorfismus (a kde je obsažena alespoň jedna virtuální členská funkce), je zahrnut ukazatel do tabulky VMT (*Virtual Method Table*). Tuto tabulku si vytváří překladač, slouží pro uchování adres všech virtuálních metod, které se vyskytují v dané třídě. Pro všechny možné instance je tato tabulka společná. Přítomnost této tabulky má vliv na celkovou velikost instance třídy, která používá mechaniky polymorfismu. Volání virtuálních metod nese rovněž riziko zvýšení časové složitosti při vykonávání funkce, neboť je předem nutné zjistit adresu volané metody *[4, str. 70-71]*.

1.3.1 Mnohotvárnost a odvozování tříd

Odvozená třída v objektově orientovaných programovacích jazycích a mechanismy spojené s dědičností (*inheritance*) vyjadřují hierarchické vztahy – podobnost – mezi třídami. Pro tento účel lze uvážit příkladné třídy *Tvar*, *Kruh* a *Trojúhelník*. Díky vypozorování vztahu „je“ u tříd *Kruh* a *Trojúhelník* v poměru ke třídě *Tvar* lze vyhodnotit, že třídy *Kruh* a *Trojúhelník* sdílí vlastnosti třídy *Tvar* (o kruhu i trojúhelníku platí, že jsou zároveň tvarem) a že se proto jedná o odvozené třídy, které vznikají dle principů dědičnosti ze základní třídy *Tvar*. Odvozené třídy poté zahrnují vlastnosti (specificky členské atributy a členské funkce) rodičovské třídy a dále ji rozšiřují o doplňující členy *[1, str. 578-579]*.

Klíčovou vlastností dědičnosti je dříve zmíněná kompatibilita mezi ukazateli rodičovské a odvozené třídy – všichni potomci jsou považováni nejen za objekty definovaného datového typu, ale také za objekty datových typů rodičů. Při dědění nejsou zahrnuty přátelské funkce ani vnořené typy, které obsahuje rodičovská třída – odvozená třída musí také deklarovat přátelskou funkci, pokud ji potřebuje použít. Přístupnost je zaručena použitím klíčového slova *protected*, které upravuje obor viditelnosti pro odvozené třídy *[4, str. 14-18]*.

Odvozování tříd má rovněž vliv na chování konstruktorů a destruktorů – v případě tvorby dědící třídy je při vytváření instance nejdříve volán konstruktor rodičovské třídy, poté potomka. Postup volání destruktoru je opačný, nejdříve je zavolán destruktor odvozené třídy, až poté destruktor třídy rodičovské. Počet vytvářených vrstev odvozených tříd (potomků) není přímo omezen, rodičovská třída *B* určité odvozené třídy *C* může být také odvozena od rodičovské třídy *A* apod. Takto lze volat celou hierarchii konstruktorů a destruktorů tříd, kdy v případě konstruktorů odpovídá pořadí od obecnější třídy směrem ke specifičtější, u destruktorů opačně – od specifičtější zpět k obecnější. Pokud dopředu předpokládáme, že odvozená třída obsahující destruktor bude sloužit jako rodičovská třída, pak je vhodné označit destruktor jako virtuální. Touto praktikou lze předcházet tvorbě statických vazeb překladačem, u kterých může, v případě využití ukazatele nebo referenční proměnné na rodičovskou třídu odkazující na odvozenou třídu, dojít ke kolizi *[5, str. 906-940]*.

V mnoha případech praktického užití dědičnosti nelze správně definovat objekt pomocí vztahu „je“ – nelze například prohlásit, že pomyslná třída *Auto* je třída *Kolo*, i když potřebujeme, aby společně tvořily celek. V takových případech může být vztah typu „má“, poté to tedy znamená, že třída v sobě zahrnuje jiný objekt a vyjadřuje skládání tříd. Nově složená třída není potomkem jiné třídy, ale využívá služeb této třídy, při příkladném skládání pak třída *Auto* využívá služeb třídy *Kolo [4, str. 45-46]*.

Dědičnost je po definici přátelských funkcí druhou nejpevnější vazbou mezi objekty v programovacím jazyce C++. Pevné vztahové vazby mezi objekty jsou nežádoucí, pokud možno, je vhodné se jim vyhnout. V případě dosažení potřebné funkcionality za pomocí skládání (kompozice) namísto dědičnosti lze očekávat vyšší efektivitu návrhu programu, obecně díky omezení tvorby pevných vazeb (kde to je možné). Je-li využita kompozice, pak mohou být uchovány a využity zahrnuté objekty s vyšší kontrolou pevnosti vazby. Ke kladným vlastnostem skládání oproti dědičnosti patří vyšší flexibilita bez narušení obsluhy volání, širší použitelnost (v případě, že je objekt přímým členem třídy), vyšší robustnost a zjednodušení návrhu *[6, str. 73-74]*.

1.3.2 Dodatek k virtuálním funkcím

Virtuální funkce je členská funkce, která je dynamicky vázána na obsluhu volání funkcí. V C++ je obsluha volání funkcí svázána (*binding*) s funkcí, která je vybrána na základě kontextu. Je-li výběr specifické funkce prováděn v době překladu programu, pak se jedná o statické vázání (*static binding*), jinak označované jako „časná vazba“. Pokud je však stejná funkce označena klíčovým slovem *virtual*, pak je provedeno vázání dynamické – rozhodnutí o výběru funkce je prováděno v běhu programu dle typu objektu, který je zodpovědný za volání funkce – tento proces je označován jako „pozdní vazba“. Klíčové slovo se umisťuje v případě deklarace nebo prototypu funkce. Bude-li členská funkce označena jako virtuální v rodičovské třídě, pak předefinované verze téže funkce, které se vyskytují v odvozených třídách, automaticky získají příznak klíčového slova *virtual* a není nutné je znovu tímto způsobem označovat, lze tak učinit pouze na základě zajištění přehlednosti *[5, str. 931-932]*.

Virtuální funkce v odvozené třídě musí mít shodný název, počet a typ přijímacích parametrů jako u předka třídy. Virtuálními metodami se nemohou stát konstruktory ani statické členské funkce (nemusí u nich totiž ani existovat instance, není nutné uvažovat o změně za běhu programu) *[4, str. 63]*.

Virtuální funkce není vhodné používat v konstruktorech či destruktorech tříd, zejména kvůli riziku nepředvídatelného chování programu při přímém či nepřímém volání neimplementované čistě virtuální funkce. V případě nutnosti implementace virtuální operace pro odvozenou třídu z konstruktoru či destruktoru rodičovské třídy je vhodné použít jiné techniky, jako například post-konstruktory – volat virtuální členské funkce, které jsou zpracovány ihned po dokončení tvorby objektu konstruktorem *[6, str. 102]*.

1.4 Osvojení základů algoritmizace

Kromě témat spojených přímo s procvičováním programovacího jazyka C++ v rámci výukového kurzu jsem implementoval také úkol se zaměřením na řešení algoritmizačního problému. Pro odlišení od častěji uváděných řadicích algoritmů, u kterých je kladen důraz na výběr typu dle časové složitosti vykonání, je tématem úlohy obdoba řešení algoritmizačního problému Jezdcovy procházky (*Knight's Tour*). Student využívá dosud procvičovaných

praktik spojených s prací se třídami a jejich členy, rovněž navrhuje členské funkce pro ovládání pohybu šachové figurky jezdce v rámci šachovnice dle pravidel problematiky, může se blíže seznámit s využitím rekurzivní funkce.

1.4.1 Pravidla Jezdcovy procházky

Základem Jezdcovy procházky je ověření existence cesty při navštívení všech polí šachovnice pomocí šachové figurky jezdce, která se může pohybovat mezi jednotlivými poli pouze ve směru odpovídajícím podobě písmene „L“, tj. o dva řádky a jeden sloupec či o dva sloupce a jeden řádek ve směru světových stran. Předpokladem validního absolvování cesty napříč šachovnicí je navštívení všech polí pouze jedenkrát a v mezích stanovených rozměrů hrací desky. Šachovnice mohou být čtvercové ($n \times n$ polí) nebo obdélníkové ($m \times n$ polí), v modifikacích problematiky lze uvažovat plochy i třírozměrné či vícerozměrné. Nalezení cesty ovlivňují nejen rozměry šachovnice, ale také výchozí pozice figurky a způsob implementace vyhledávacího algoritmu *[9]*.

Při výběru možných cest je nutné uvážit dělení na cesty otevřené (*open tours*) a uzavřené (*closed tours*). Toto dělení vyplývá z předpokladu o cílové pozici šachové figurky – v případě otevřených cest lze dosáhnout všech pozic v rámci šachovnice bez návratu na výchozí pozici (konečný bod se nerovná počátečnímu). U uzavřených cest jsou úspěšně navštívena všechna pole a na rozdíl od předchozí varianty současně platí, že by poslední navštívené pole umožnilo přechod do počáteční pozice, pokud by tedy nebylo na začátku cesty označeno jako navštívené. Formálně lze otevřené cesty specifikovat jako cesty hamiltonovské, ty uzavřené pak jako hamiltonovské kružnice (resp. hamiltonovské cykly) *[10, str. 251]*.

1.4.2 Vyhledání otevřených či uzavřených cest

Dle historických výzkumů problematiky Jezdcovy procházky bylo potvrzeno či vyvráceno více teorií spojených se zobecněním řešení pro nalezení otevřených nebo uzavřených cest. Řešení pro zjednodušení může spočívat v rozdělení čtvercové šachovnice na poloviny, nebo na více menších čtvercových šachovnic, které lze snáze řešit jednotlivě – zde platí, že před vstupem do další části hrací desky je nutné navštívit všechna pole předchozí části. U čtvercových šachovnic s počtem řádků a sloupců menším než 5 zásadně nelze docílit úspěšné otevřené ani uzavřené cesty (u hracích ploch s rozměry 4×4 nebo 3×3 polí vždy zbývá minimálně jedno nenavštívené pole, u menších pak není možný jediný pohyb) *[10, str. 252]*.

K vyhledávání cest lze z hlediska implementace přistupovat více možnými způsoby – každá výchozí pozice figurky může nabízet maximálně 8 možných tahů, přičemž je třeba uvážit, jsou-li následující pozice uvnitř hranic hrací plochy. Pozice mimo oblast šachovnice nelze považovat za validní pole pro přesun jezdce. U pozic, které nejsou počáteční (aktuální cesta probíhá), kontrolujeme kromě hranic šachovnice také skutečnost, zdali nebyly potenciální nové pozice figurky dříve navštíveny. Mezi validními tahy lze vybírat buďto systematicky (tedy například po směru hodinových ručiček) nebo nahodile. Tímto způsobem je realizován přesun jezdce v rámci šachovnice do doby, dokud nejsou všechna pole navštívena. Výběrem polí lze ovšem docílit předčasného ukončení cesty z důvodu absence dalších možných tahů i v případě, že existuje platné řešení pro úspěšné obsazení všech pozic. Tehdy je možné využít navrácení na předcházející pozici a zvolit jinou cestu – tato metoda je označována jako *backtracking.* Návraty figurky jezdce lze řídit uchováním navštívených pozic, například pomocí datové struktury zásobníku, funkce pro přesun může být řešena iterativní i rekurzivní metodou (funkce volá sebe sama). Navrácení je možné provádět až po výchozí pozici šachové figurky jezdce *[11]*.

1.5 Práce s knihovnami v jazyce C++

Mnoho programů vytvořených v jazyce C++ spoléhá na využití knihoven, které do nich lze zahrnout. Jejich účelem je znovupoužití již vytvořených součástí, které slouží pro usnadnění vývoje, ať už z hlediska dostupnosti datových typů či struktur, nebo samotné údržby. Základní knihovna jazyka C++ (Standard-Library) nabízí například nástroje pro práci s řetězci nebo vstupně výstupními proudy, datovými kontejnery a algoritmy, regulárními výrazy nebo s režimem provozu na více vláknech. Myšlenkou tvorby knihoven je zjednodušení programování, nabídnutí optimálního řešení vybrané problematiky a také umožnění naučení ověřených principů *[1, str. 87-89]*.

1.5.1 Využití typu string

Na rozdíl od definice textových řetězců ve formě znakových polí ukončených nulovým znakem, které jsou charakteristické pro programovací jazyk C, je pro tento účel v rámci standardní knihovny dostupný typ *string*. Kromě mnoha dalších možností nabízí tento typ například snadné slučování řetězců (prostřednictvím znaménka +) nebo rozdělení na více částí

dle specifikovaných kritérií ve formě počátečního indexu písmene a délky části řetězce, jiným příkladem manipulace s řetězci může být nahrazování jeho části jinou posloupností znaků. Textové řetězce, které jsou ukládány pomocí datového typu *string*, lze vzájemně porovnávat přímo pomocí operátoru dvou znaků rovnosti (==) *[1, str. 90-91]*.

U definice řetězců ve stylu jazyka C se můžeme setkat s úskalími, která typ *string* vyřeší za nás. Jedním z příkladů je uvolnění alokované paměti, které je v případě využití typu *string* zajištěno automaticky. Základním rozdílem mezi oběma způsoby zápisu textových řetězců je skutečnost, že *string* je řádný typ s konvenční sémantikou, zatímco zápis řetězců dle jazyka C přestavuje sadu konvencí, která je podpořena užitečnými funkcemi *[1, str. 1038]*.

Některé funkce knihovny jazyka C++ neumožňují přijetí argumentů ve formě typu *string* – v rané fázi vývoje programovacího jazyka nebyl ještě dostupný, proto je třeba předat řetězec ve stylu jazyka C (příkladem může být funkce pro specifikaci názvu při otevírání souboru). Rovněž je třeba předpokládat, že importované knihovny programované v jazyce C také přijímají tuto původní podobu řetězců *[5, str. 556]*.

1.5.2 Knihovna STL

Standardní knihovna šablon jazyka C++ (Standard Template Library) zahrnuje šablony různých typů pro využití algoritmů a datových struktur (kontejnerů a jejich iterátorů). Tyto generické šablony lze využít pro implementaci abstraktních datových typů. Pojem kontejner označuje třídu, která ukládá data a umožňuje nad nimi provádět operace. Tyto datové struktury se dělí na dva typy – sekvenční (ukládání dat prováděno v určité posloupnosti obdobně jako pole) a asociativní (ukládání dat probíhá na základě klíče, který specifikuje pozici). Příkladem sekvenčních datových typů je dynamicky rozšiřitelné pole (vector), lineární seznam (list) nebo obousměrný seznam (deque). V případě asociačních struktur lze zmínit sadu unikátních klíčů (set), sadu opakovatelných klíčů (multiset), hashovací tabulku unikátních klíčů (map) nebo hashovací tabulku opakovatelných klíčů (multimap) *[5, str. 1006]*.

Pro přístup k datům uchovaným v kontejnerech se využívá iterátorů, které se podobají ukazatelům. Iterátory jsou si rovny, pokud ukazují na stejnou hodnotu v rámci struktury. Jejich posun je realizován operátory „++“ a „--“, obdobně jako u zvyšování a snižování číselných hodnot. Podoba operací s iterátorem je vázána na konkrétní datovou strukturu, tak lze odlišit iterátory s povoleným posunem v jednom směru, v obou směrech, náhodně apod. *[4, str. 240]*.

Dynamické pole neboli *vector* patří k nejpoužívanějším kontejnerům, se kterými se lze setkat nejen v programovacím jazyce C++. Dynamické pole je specifické sekvenčním ukládáním dat vybraného typu, jeho velikost automaticky narůstá dle potřeby. Aktuální velikost pole (tj. počet existujících položek) lze zjistit zavoláním členské funkce *size*. Velikost lze explicitně nastavit při inicializaci dynamického pole předáním argumentu v kulatých závorkách. Nové položky je možné přidat voláním členské funkce *push_back*, která připojí novou položku na konec pole a zvýší velikost o 1. Ukládat lze zabudované číselné typy, uživatelem definované typy i ukazatele *[1, str. 95-97]*.

Omezené možnosti polí, která nejsou dynamická, můžou vést k řadám programovacích chyb (nejedná-li se o případ, kdy známe pevně danou velikost již v době překladu). K chybám dochází nejčastěji v souvislosti s využíváním polí pevné velikosti a ukazatelové aritmetiky ve stylu jazyka C v případech manuální správy paměti bez využití lépe ovladatelných vyrovnávacích pamětí, dynamicky alokovaných polí anebo řetězců typu *string*. Výhody dynamických polí tedy spočívají především v automatické správě paměti, v bohatém rozhraní (pomocí kterého lze snáze implementovat i složité úkony) a ve zpětné kompatibilitě typu *string* a struktur jako dynamické pole s jazykem C. Žádoucí vlastnosti se přitom na ztrátě výkonu projeví pouze zanedbatelně *[6, str. 166]*.

Šablonu seznamu lze zahrnout pomocí direktivy *include* a hlavičkového souboru *list*. Pomocí tohoto kontejneru lze v řadě vkládat data typu *T*. Tato šablona má k dispozici přiřazovací operátor, který umožňuje přenos obsahu jednoho seznamu do jiného. Obdobně jako u datové struktury dynamického pole z knihovny STL, je tato šablona opatřena funkcemi například pro zjišťování délky, třídění, vkládání či odstranění obsažených prvků *[4, str. 241]*.

Seznam je obousměrný, lze jej využít v případech pro ukládání po sobě jdoucích prvků bez nutnosti přesunu prvků, které jsou v seznamu již přítomny. Oproti typu vyhledávání v dynamickém poli, je u seznamu využíváno průchodu touto strukturou s hledáním prvku dle předané hodnoty, tento sekvenční průchod buďto nachází hledaný element, nebo neúspěšně dosahuje konce seznamu. Pro identifikaci konkrétního prvku v seznamu můžeme opět využít iterátoru. Šablonu seznamu je v porovnání s dynamickým polem vhodnější použít v případě většího množství navazujících prvků. Šablona *vector* však nabízí vyšší výkon při práci s operacemi, kam spadají vyhledávací a řadicí algoritmy *[1, str. 98-99]*.

1.5.3 Chytré ukazatele

Standard C++ 11 rozšiřuje nabídku ukazatelů, které má programátor k dispozici, o tzv. chytré ukazatele (*smart pointers*). Jedná se o objekty, které zajišťují funkcionalitu běžných ukazatelů, dále však mohou automaticky uvolnit alokovanou paměť, která už není využívána – tato skutečnost pomáhá zabránit výskytu paměťových úniků. Mezi nejpoužívanější kandidáty chytrých ukazatelů patří *unique_ptr*, *shared_ptr* a *weak_ptr*. Zmíněné chytré ukazatele jsou programátorovi k dispozici po zahrnutí hlavičkového souboru *memory* pomocí direktivy *include.* Tvorba chytrého ukazatele se liší od způsobu vytváření běžného ukazatele – po výběru typu chytrého ukazatele je třeba do ostrých závorek uvést datový typ, na který se ukazatel vytváří (zápis poté může být např. *unique_ptr<int> ukazatel(new int);*) *[5, str. 533]*.

Rozhodnutí o využití konkrétního typu chytrého ukazatele závisí na situaci. *unique_ptr* je ukazatelem na individuální objekt (nebo pole objektů), kdežto *shared_ptr* je na rozdíl od předchozího typu chytrého ukazatele namísto přesunutí zkopírován. Tím pádem ukazatele na objekt typu *shared_ptr* sdílí vlastnictví objektu, tento objekt je z paměti uvolněn až při odstranění posledního z existujících *shared_ptr* ukazatelů. U tohoto typu chytrého ukazatele se však lze setkat s určitou režií z hlediska optimalizace, proto je vhodné *shared_ptr* využít pouze v případě nutného sdílení vlastnictví. Potřebujeme-li se odkázat na polymorfní objekt, běžně využíváme *unique_ptr*. Pokud je třeba objekt sdílet (a není zřejmý pouze jediný vlastník objektu), pak využíváme *shared_ptr [1, str. 112-114]*.

Chytrý ukazatel *weak_ptr* odkazuje na objekt, který je spravován ukazatelem *shared_ptr*. K vlastnostem *weak_ptr* patří možnost přístupu k objektu pouze v případě, že tento objekt existuje. U tohoto objektu lze také očekávat, že může být kdykoliv odstraněn. Nejčastějšího využití nachází *weak_ptr* při ukončení průběhu cyklů v datových strukturách, které jsou spravovány za pomoci *shared_ptr* ukazatelů *[1, str. 993]*.

1.5.4 Práce s knihovnou SQLite

SQLite představuje jednoduchou a snadno implementovatelnou komponentu pro řízení báze dat, která nevyžaduje složitý instalační proces, spoléhá pouze na využití překladače jazyka C/C++ a zajišťuje funkcionalitu na operačních systémech Windows, Linux i Mac OS X. Knihovna SQLite nevyžaduje správu obsluhy požadavků na server, lze ji přímo zahrnout do

(lokálně) vyvíjené aplikace. Ačkoliv je primárním účelem této knihovny jednoduchost a některé funkce mohou postrádat zcela optimální způsob řešení, lze aplikovat základní příkazy jazyka SQL, které slouží pro tvorbu tabulek a jejich manipulaci ve formě správy obsahu, řízení vztahů mezi tabulkami, realizaci spouštěčů (triggerů) a podobně *[12, str. 2-3]*.

K omezením SQLite vyplývajícím z jednoduchosti patří možnost vykonávání pouze ‚plochých' transakcí (tj. není možné aplikovat techniky vnořování transakcí, které by umožnily provádět transakci uvnitř již probíhající transakce), rovněž není možné ukládat předcházející stav databáze, který by bylo možné v případě nutnosti obnovit. SQLite také nemůže nabídnout velkou podporu zpracování na více vláknech, nikoliv z hlediska počtu probíhajících operací pro čtení z databáze, ale při zápisu do jednoho souboru databáze (může být vždy jen jeden probíhající zápis) *[12, str. 16-17]*.

Před využitím možností SQLite je třeba získat potřebné implementační a hlavičkové soubory ve zdrojové či binární formě. Do existující aplikace v programovacím jazyce C nebo C++ lze importovat soubor *sqlite3.h*, a to pomocí direktivy *include*. Zahrnutím zdrojových souborů se programátorovi zpřístupní funkce (vytvořené v programovacím jazyce C) pro obsluhu práce s bází dat. Pro otevření existující (případně vytvoření nové) databáze lze využít funkci *sqlite3_open*, která přijímá prvním argumentem název souboru databáze, druhým argumentem ukazatel na typ *sqlite3*, jež je potřebný pro provedení dalších operací nad tímto souborem. Výstupem funkce *sqlite3_open* je číselná hodnota, která určuje úspešnost operace otevření. Takto otevřený soubor databáze zůstává za běhu programu přístupný dalším operacím do doby, dokud není správně uzavřen pomocí funkce *sqlite3_close*. Vykonání funkce uzavření má za následek zrušení vytvořeného spojení s databází a uvolnění všech alokovaných zdrojů. Vstupním argumentem je ukazatel na dříve vytvořený typ *sqlite3,* který se úspěšným uzavřením databáze stává neplatným *[12, str. 5-8]*.

Vykonání transakčních operací nad databází je spojeno s využitím jazyka SQL, vytvářené příkazy jsou uchovány ve formě textového řetězce. Je-li aktivně otevřena databáze, pak je možné vykonat funkci *sqlite3_exec,* které se pomocí argumentu předá ukazatel na vytvořený typ databáze *sqlite3* a znění SQL příkazu v podobě textového řetězce. Touto přímou metodou vykonání transakce lze volat všechny příkazy jazyka SQL pomocí funkce *sqlite3_exec,* bez ohledu na charakter transakční operace (což zahrnuje mimo jiné vytváření tabulky, vkládání položek do tabulky, výpis položek z tabulky apod.) *[12, str. 9-10]*.

1.6 Využití principu buněčného automatu

Buněčné (celulární) automaty popisují změnu stavu prvků (resp. buněk) na základě svého okolí. Pozorovatelný vývoj probíhá v diskrétním čase, přičemž stav buňky nové iterace (časové změny) závisí na evolučním vzoru přechodu mezi iteracemi a okolním prostředí buněk. Buněčný automat lze formálně vyjádřit množinou ($\mathbb{Z}^n$, S, N, f), kde $\mathbb{Z}$ vyjadřuje celé číslo rozměru buněčného pole, n vyjadřuje dimenzi (vyšší než 1), celkový konečný počet stavů S, sousedství buňky N (které je množinou prvků $\mathbb{Z}^n$) a f představující přechodovou funkci. Nejběžnějším zobrazením buněčného prostoru je dvourozměrné pole, tj. matice ($\mathbb{Z}^2$), kde je každá buňka reprezentována čtvercem v poli, které má podobu šachovnice. Při této konfiguraci má každá buňka maximálně osm sousedů. Okolní prostředí buněk lze v případě dvourozměrného pole dělit na *von Neumannovo*, kde jsou vyhodnoceny čtyři okolní buňky, které svou pozicí odpovídají směru světových stran, dále na sousedství hexagonální (které navíc zahrnuje dvě buňky nacházející se na jedné z diagonál) vycházející z šestiúhelníkové základny pole a na *Mooreovo* okolní prostředí, ve kterém se zohledňují všechna okolní pole buňky (kterých je v případě dvourozměrného zobrazení s čtvercovou základnou osm) *[13]*.

Při implementaci úlohy hry života se využívá právě principů buněčných automatů. Buněčný prostor je tvořen dvourozměrným nekonečným polem, jehož buňky mají čtvercovou základnu, tím tedy maximálně 8 sousedů. Každá z buněk v matici se může nacházet v jednom ze dvou stavů – může být živá nebo mrtvá. Právě stav dané buňky ovlivňuje chování buněk, které se nachází v jejím okolním prostředí (využívá se okolí *Mooreova*). Pro transformaci mezi generacemi (v diskrétním čase) se využívá přechodové funkce *B3/S23*, která simuluje životní proces a vymezuje pravidla hry. Každá živá buňka, která má méně než dva živé sousedy, umírá. Rovněž umírá každá živá buňka, která má více než tři živé sousedy (dále přežívá tedy buňka, která má právě dva nebo tři živé sousedy). Kterákoliv mrtvá buňka, jež má přesně tři živé sousedy, v další generaci ožívá. První generace vzniká aplikací těchto pravidel současně na každou buňku v prostoru, vychází se ze stavu předchozí (v tomto případě nulté) generace a nově provedené změny části matice vykonané přechodem mezi generacemi se při kontrole okolí buněk neuvažují. Iterační proces tvorby nových generací má za následek tvorbu shluků živých buněk, kterým se říká vzory (*patterns*) a které se člení na statické prvky (skupiny žijících buněk, u kterých procesem stárnutí samovolně nedochází ke změně stavu), oscilátory (věčně trvající pohyblivé vzory, které se s danou periodou opakují) a pohyblivé skupiny buněk, které jsou označovány jako lodě (*spaceships*). Ačkoliv jsou dle základních pravidel hry života hranice dvourozměrného pole neomezené, konstrukční prostředky při

programování nikoliv. Dosažení deklarovaných hranic matice lze z hlediska programování řešit méně sofistikovaným předpokladem, že všechny buňky nacházející se mimo hranice dvourozměrného pole jsou mrtvé (čímž je překročení hranic znemožněno), nebo lze implementovat návaznost horní a spodní, stejně tak levé a pravé části matice. Touto technikou pak není buněčný prostor omezen hranicemi a buňky, které z jedné strany prostor opustí, pokračují v trajektorii ze strany protější *[14]*.

2 PODKLAD PRO NÁVRH TESTOVACÍCH SCÉNÁŘŮ A JEDNOTKOVÝCH TESTŮ

Jednotkové testy (*unit tests*) slouží pro kontrolu modulů, tj. samostatných jednotek, které jsou obsaženy v projektu s úkolem. Testovací scénáře pokrývají očekávanou funkcionalitu testované součásti, jsou složeny z dílčích testovacích případů a jsou význačné svým předpokládaným výstupem. Úlohy programování v jazyce C++ obsahují tyto scénáře pro vyhodnocení úspěšnosti implementace členských funkcí tříd, které implementuje student.

2.1 Postup návrhu testovacích scénářů

V závislosti na vybrané úloze a funkcionalitě, které je třeba docílit, se definují testovací sady. Ty obsahují případy pro přiřazení očekávaných hodnot k výstupům členských funkcí a jejich následnému porovnání. Právě shoda porovnávaných prvků (je-li očekávána rovnost) napovídá, že bylo docíleno úspěšné implementace testované funkce.

2.1.1 Testování formou jednotkových testů

Jednotkové testy spočívají v testování modulů. Obecně zahrnují testování subprogramů, subrutin či procedur (funkcí) – testování je zaměřeno na malé stavební bloky programu. Cílem je porovnání funkcionality daného modulu se stanoveným požadavkem, který se k testovanému modulu váže. Jednotkové testování se silně pojí s testovací technikou *bílé skříňky* (*white-box*), kdy je k dispozici zdrojový kód vytvořených modulů. K přínosům testování malých částí programu patří nezávislé ověření jinak propojených komponent a také paralelní testování většího množství modulů současně *[15, str. 70]*.

Jednotkový test se skládá z testovacích případů. Každý testovací případ má potvrdit správnost implementace pro konkrétní okolnosti, tj. vstupní data a vnitřní stavy programu. Vytvořené testy zvyšují kvalitu výsledného produktu zajištěním vyšší robustnosti a spolehlivosti, poskytují okamžitou motivaci k úpravě chybně implementovaných součástí. K testování je třeba přistoupit tak, aby bylo chyb nalezeno co nejvíce, což je vzhledem k povaze procesu tvorby testů složité (zejména, je-li testerem samotný programátor). Testy lze rozlišit jako úspěšné v případě nálezu chybně implementované části kódu, neúspěšné v situaci špatného návrhu pokrytí testovaných součástí *[15, str. 10-11]*.

Testy jsou typicky psány ve stejném programovacím jazyce jako testovaný program. Každý jednotkový test i testovací případ má být pojmenován názvem, který jej výstižně popisuje. Test je obvykle strukturován do čtyř částí obsahující příkazy pro nastavení kontextu k vykonání operace nad testovaným modulem (nepovinná součást), dále je zahrnut jeden či více příkazů pro vyvolání chování funkce, které je ověřováno. Třetí část obsahuje alespoň jeden příkaz pro ověření výstupu testu s očekávanou hodnotou, poslední část (která je rovněž v závislosti na použití nepovinná) spočívá v závěrečném „úklidu“, například v uvolnění alokované paměti *[16, str. 55-56]*.

Při návrhu testů je třeba uvážit jak správné či předpokládané vstupní podmínky testovaných modulů, tak i ty naopak neplatné nebo neočekávané. U tvorby testů běžně dochází k zaměření na platné a očekávané vstupní okolnosti (které jsou často v očekávaném výstupu funkce obsaženy jako jediné), přičemž ty nesprávné a neočekávané zůstávají opomenuty. Oproti uvádění očekávaných hodnot, které jsou porovnány s výstupem testovaného modulu, odhalí nežádoucí a neočekávané vstupní podmínky mnohdy větší množství defektů *[15, str. 18]*.

2.1.2 Kritéria testovacích scénářů

Při testování programu, resp. dílčího modulu, se ověřuje malá podmnožina všech možných vstupů. Správný výběr podmnožiny směřuje k nalezení co nejvyššího počtu chyb. Každý testovací případ by měl maximálně pokrýt možné vstupní hodnoty (což vede ke snížení nutného počtu vytvořených testovacích případů) a vede k rozdělení všech možných vstupů do konečného počtu ekvivalentních tříd – pokud testovaná hodnota z třídy nachází chybu, je předpokladem, že všechny ostatní testovací případy téže třídy dosahují nálezu stejné chyby. Třídy ekvivalence jsou rozlišovány jako platné a neplatné. Příkladem může být předpoklad platné celočíselné hodnoty od 1 do 999 (tento rozsah, včetně hraničních hodnot 1 a 999, je interpretován platnou ekvivalentní třídou), jak čísla s hodnotou 0 a méně, tak čísla 1000 a vyšší pak spadají do neplatné ekvivalentní třídy *[15, str. 42-43]*.

Zmíněné hraniční hodnoty zahrnují prvky na hraně ekvivalentních tříd, stejně tak nejbližší menší nebo větší prvek vstupních a výstupních tříd ekvivalence. Pokud je tedy specifikována množina vstupních číselných hodnot, je vhodné utvářet testovací případy potvrzující platnost hodnot z obou konců rozsahu množiny, neplatnost u hodnot těsně přesahující hranice ekvivalentní třídy. V případě předpokládaných celočíselných hodnot od 1 do 255, kdy hodnoty 1 i 255 spadají do platného rozsahu, je cílem testovat hraniční hodnoty 0 (neplatná), 1 (platná), 255 (platná) a 256 (neplatná) *[15, str. 46]*.

Při testování úloh se vychází také z dokumentace, která popisuje předpokládanou funkcionalitu a pomáhá stanovit obsah testovacích případů. Dokumentace slouží jako podklad pro tvorbu scénářů, zpětně však i samotná dokumentace musí být předmětem kontroly (pro zajištění přesnosti a jasného vyjádření očekávané činnosti). Všechny příkladné situace, které jsou v dokumentaci vytyčeny, by měly být vyjádřeny testovacím případem *[15, str. 103]*.

2.1.3 Testy řízený vývoj aplikace

Oproti tvorbě jednotkových testů na základě existující implementace programu je technika testy řízeného vývoje TDD (*Test-driven Development*) stručně vyjádřený proces, který do tvorby softwaru jednotkové testy přímo začleňuje. Tvorba testů předchází vývoji programového řešení aplikace, navržené testovací případy jsou konzistentní a nabízí možnosti bezpečné úpravy existujícího zpracování v kódu. Implementace nového chování aplikace je podmíněna tvorbou testovacího případu, který toto chování definuje, neúspěch navrženého testu naznačuje potřebnou změnu v programu. Při testy řízeném vývoji je očekáváno pozvolné navržení jednoduchých testů, které vyjadřují nejmenší možné doplnění programu. Tato technika vývoje staví na opakování cyklu navržení jednoduchého testu, dále ověření, že navržený test není úspěšný (z důvodu dosud chybějící implementace), tvorbě kódu pro zajištění funkcionality definované testem a na případné estetické úpravě nově implementovaných součástí *[16, str. 56-59]*.

2.2 Využití knihovny MinUnit

Pro realizaci navržených testovacích scénářů ve formě příslušných testovacích případů lze využít knihovnu MinUnit. Jedná se o nerozsáhlý nástroj pro tvorbu jednotkových testů v programovacím jazyce C/C++, který poskytuje možnost definice a konfigurace testovacích sad pomocí vybraných přiřazovacích typů. Nástroj zajišťuje výpis informací o počtu spuštěných testů, provedených přiřazení a o celkové době potřebné pro vykonání.

MinUnit je obsažen v jediném hlavičkovém souboru, který musí být pro použití v programu zahrnut pomocí direktivy *include*. Funkce pro spuštění sady testovacích případů *MU_RUN_SUITE* je definována pomocí makra, které je třeba vstupním argumentem funkce předat název testovací sady pro vykonání. Testovací sadu lze definovat makrem *MU_TEST_SUITE*, které se vstupním argumentem funkce určí její název. Sada testovacích

případů obsahuje uvedení makra *MU_RUN_TEST* zajišťující volání testovacího případu, který je definován názvem předaným ve vstupním argumentu funkce. Prvek *MU_TEST*, jež je rovněž definován makrem v hlavičkovém souboru nástroje MinUnit, má ve vstupním argumentu funkce určen svůj název a obsahuje přiřazení, která jsou předmětem testovacího případu. Po vyvolání *MU_RUN_SUITE* v rámci hlavní smyčky programu je využito funkce *MU_REPORT* pro zobrazení výsledků testů a potřebného času pro jejich vykonání.

Nástroj MinUnit nabízí přiřazovací funkci *mu_check,* jejímž vstupním argumentem je podmínka, která v případě navrácení pravdivostní hodnoty *true* vede k úspěchu přiřazení. Neúspěch přiřazení funkcí *mu_check* vyvolává chybovou zprávu. Funkce *mu_fail* způsobí selhání testu a zobrazí textový řetězec se zprávou, která byla zadána vstupním argumentem této funkce. Pro ověření platnosti podmínky s výpisem vlastní chybové zprávy lze využít funkce *mu_assert*, která argumentem přijímá podmínku pro vyhodnocení a také znění zprávy, jež se v případě neúspěchu (když podmínka navrací pravdivostní hodnotu *false*) vypisuje. Funkce pro porovnání hodnot celých čísel *mu_assert_int_eq* argumentem přijímá očekávanou a výslednou hodnotu, rovnost předaných hodnot zajistí úspěch přiřazení, nerovnost vyvolá chybovou hlášku s uvedením obou hodnot. Funkce *mu_assert_double_eq* namísto celých čísel porovnává hodnoty z oboru reálných čísel dle určité přesnosti, kterou lze stanovit konfigurací definované hodnoty *MINUNIT_EPSILON*. Dostatečná blízkost k rovnosti dvou předaných hodnost vede k úspěšnému provedení testu, neúspěch je význačný chybovou zprávou se zobrazením obou předaných hodnot *[17]*.

3 NÁVRH TVORBY DOKUMENTACE A FORMÁTOVÁNÍ ZDROJOVÉHO KÓDU

Dokumentace zdrojového kódu projektů s úlohami poskytuje vyšší přehlednost implementovaných součástí a zároveň nabízí objasnění definované funkcionality. Formát popisu je na rozdíl od standardních bloků s komentáři strukturován pro konkrétní modul, vycházení z dodržovaných praktik dále zvyšuje konzistenci popisu a tím pádem i celkovou čitelnost. Soubory se zdrojovým kódem úloh programování v jazyce C++ obsahují dokumentaci, která odpovídá syntaxi generátoru Doxygen.

Čitelnost zdrojového kódu ovlivňují rovněž různé styly jeho zápisu. Pro sjednocení vlastností, jako jsou například maximální délka řádku, počet mezer při odsazení bloků kódu nebo pozice závorek, lze mnohdy využít nástroje pro sjednocení formátování. K těmto nástrojům se řadí i Clang-Format, který lze integrovat napřímo do vývojového prostředí.

3.1 Dokumentace ve formátu Doxygen

Doxygen je standardním nástrojem pro tvorbu dokumentace, který je podporován širokou škálou programovacích jazyků. Nástroj jako takový slouží pro generování online dokumentace zdrojového kódu ve formátu HTML, nebo například pro tvorbu referenční příručky vybrané sady zdrojových souborů. Vztahy mezi jednotlivými součástmi programu mohou být vyjádřeny pomocí automaticky generovaných grafů, nebo třeba diagramů dědičnosti. Dokumentace v jednoduchém provedení může být utvořena snadným textovým zápisem, speciální příkazy generátoru Doxygen nebo značky HTML lze využít pro pokročilejší strukturování výstupu. Podporována je dokumentace souborů, jmenných prostorů, tříd, struktur, šablon, kromě dalších také členských vlastností a funkcí *[18]*.

Blok dokumentace ve formátu Doxygen má podobu víceřádkového komentáře ve stylu programovacího jazyka C/C++ s doplňujícími značkami, které utváří strukturovaný text pro konverzi do generovaného výstupu dokumentace. Pro každý prvek zdrojového kódu (např. členskou vlastnost či funkci) existují dva až tři způsoby zápisu – stručný a podrobný. Třetím způsobem je popis vložený, který lze využít například u členských funkcí, tento režim sjednocuje bloky komentářů obsažených uvnitř funkce. Stručný popis je význačný jednořádkovým zápisem, zatímco podrobný poskytuje delší a rozsáhlejší informace o daném modulu.

Kromě umístění bloku s dokumentací před deklaraci prvku v hlavičkovém souboru podporuje Doxygen vkládání popisu i před jeho definici (dokumentace je tak obsažena v souboru implementačním). Variantou je rovněž umístění stručného popisu před deklaraci a detailního popisu až před definici prvku s možností rozdělení do více souborů. Při zájmu o umístění popisu za členem, namísto před ním, je třeba doplnit blok s dokumentací o znak ostré závorky vlevo (<) *[19]*.

Značky a speciální příkazy, které je možné využít pro rozlišení obsahu popisovaných součástí v bloku dokumentace, lze dle vybraného stylu uvádět za znakem zpětného lomítka (\) nebo zavináče (@). K běžným značkám patří *brief* pro stručný popis dokumentované součásti, *param* pro zaznamenání parametrů, které lze dále rozlišit na vstupní, výstupní a vstupně-výstupní, nebo *return* či *returns*, jež označují popis návratové hodnoty. Značka *class* se uvádí v případě deklarace třídy, za touto značkou následuje její název. Dokumentační bloky se utváří nejen pro jednotlivé členy, ale také pro jednotlivé soubory ve formě záhlaví. Blok pro popis souboru může využít například značky *file* pro uvedení názvu souboru (včetně koncovky), *author* nebo *authors* při zaznamenání autora, *date* pro uchování data vytvoření či úpravy, *version* k uchování textového řetězce současné verze souboru nebo třeba značky *par* pro vytvoření samostatného odstavce *[20]*.

3.2 Formátování kódu dle Clang-Format

Nástroje Clang využívají rozhraní příkazového řádku pro rychlou kontrolu syntaxe a, mimo jiné, také pro automatické formátování zdrojového kódu. Jedním z hlavních nástrojů je i Clang-Format, který tvoří knihovnu a samostatný formátovací prostředek, s cílem automatického formátování zdrojových souborů (nejen) programovacího jazyka C++ dle nastavitelných stylů. Clang-Format funguje na principu převodu vstupního souboru do proudu tokenů (příznaků), kolem kterých jsou dle stylu upraveny bílé znaky (mezery, tabulátory, prázdné znaky či konce řádků), jež se v kódu vyskytují. Nástroj byl vytvořen pro využití formou integrace do vývojových prostředí a také jako součást jiných nástrojů pro úpravu stylu zdrojového kódu *[21]*.

Jednotlivé parametry pro vybraný styl lze vybírat z předem definovaných stylů (*LLVM*, *Google*, *Chromium* apod.), případně lze vlastní konfiguraci vytvořit úpravou souboru *.clang-format*. Struktura konfiguračního souboru odpovídá formátu YAML se zápisem *parametr:*

hodnota, tyto hodnoty mohou být odlišně nastaveny pro více programovacích jazyků v rámci jednoho souboru. K mnoha parametrům, které lze pozměnit, patří *ColumnLimit* pro úpravu maximální délky jednoho řádku ve zdrojovém kódu, *AllowShortBlocksOnASingleLine* pro sloučení neobsáhlých bloků kódu oddělených závorkami na jeden řádek, *BreakBeforeBraces* nabízí mnoho variant stylů při umísťování závorek (zkráceně, vždy na nový řádek, bez nebo s odsazením apod.) nebo například *IncludeBlocks*, který abecedně a prioritně seřadí seznam souborů vložených pomocí direktivy *include [22]*.

Styl Google, který popisuje míru čitelnosti, shromažďuje využívané konvence pro údržbu (a sjednocení konzistence) zdrojového kódu při programování v jazyce C++. Obsahuje návrhy pro úpravu kódu vedoucí k zjednodušení čtení na úkor nutnosti tvorby formátu kódu dle stanovených praktik, které by měly být uplatněny stejnou mírou napříč všemi zdrojovými soubory. Samotný styl doporučuje oddělení kódu do deklaračních a definičních souborů v zájmu čitelnosti a zvýšení výkonu. Všechny hlavičkové soubory by, dle stylu, měly být opatřeny blokem tvořeným direktivou *define*, jež zabrání vícenásobnému vložení při užití direktivy *include*. Styl zdůrazňuje členění do jmenných prostorů, vyzývá k inicializaci proměnné při její deklaraci, z hlediska formátování je pro odsazení doporučeno využití mezer namísto tabulátorů a udržení řádků zdrojového kódu v délce do osmdesáti znaků. Styl Google dále stanovuje formátování bloků kódu a patřičné umístění závorek při větvení programu prostřednictvím podmínek. Deklarace tříd obsahují, dle konvence, zprvu veřejné, poté chráněné a soukromé členy, sekce viditelnosti jsou odděleny prázdným řádkem *[23]*.

4 VYHODNOCENÍ SPRÁVNOSTI KÓDU V SYSTÉMU GITLAB

Systém pro správu verzí je využíván pro uchování projektů, poskytnutí přístupu k nim, udržování informací o prováděných změnách a rovněž pro vykonání automatických operací s uloženými daty (resp. zdrojovými soubory) dle konfigurace. V takovém případě je možné vytvořit pomocné konfigurační soubory s popisem operací, které jsou prováděny například při změně zdrojových souborů uložených v systému.

4.1 Prostředky pro sestavení projektů s úlohami

Sestavení projektů nejen v programovacím jazyce C/C++ je možné, nezávisle na typu překladače, realizovat pomocí systému CMake. Jedná se o multiplatformní nástroj, který lze použít současně se základním prostředím pro sestavení programu. Konfigurace projektu pro automatizované sestavení spočívá v tvorbě textového dokumentu *CMakeLists.txt*, který je standardně umístěn v kořenovém adresáři pracovního prostoru se zdrojovými soubory. CMake tak generuje základní sestavující prostředí pro následný překlad zdrojového kódu, vytvoření knihoven a spustitelných souborů. Je podporováno statické i dynamické sestavování knihoven i tvorba více sestavení v jediném pracovním adresáři. Soubor *CMakeLists.txt* sestává z posloupnosti příkazů pro konfiguraci sestavení. Předem definované příkazy umožňují navigaci a výběr souborů pro zahrnutí, vytvoření jednoho nebo více spustitelných souborů a také doplnění případných parametrů pro sestavení s ohledem na typ operačního systému *[24]*.

Jedním se základních příkazů, které se využívají v souboru *CMakeLists.txt*, je *file*. Tento příkaz slouží pro vyhledání a umožnění další manipulace se soubory. Příkaz *file* přijímá vícero vstupních argumentů dle použití, je-li vložen argument *GLOB*, pak lze po uvedení názvu proměnné vybrat jeden nebo více souborů (je tak například možné vybrat všechny hlavičkové nebo implementační soubory ve vybraném umístění). Manipulace s označenými soubory je poté možná prostřednictvím vytvořené proměnné. Cesta k souboru je relativní v poměru ke kořenovému adresáři projektu, výběr souborů užívá principu regulárních výrazů.

Pro přidání parametrů při překladu zdrojových souborů lze využít příkaz *add_definitions*. Zde uvedené příznaky jsou přidruženy k příkazové řádce překladače pro současnou a podřazené složky. Pomocí *add_definitions* lze připojit libovolný parametr, jeho původní zaměření však zahrnovalo pouze předání definic pro preprocesor.

Při práci s proměnnými v rámci konfiguračního souboru *CMakeLists.txt* je možné přiřazovat hodnoty předávané vstupním argumentem pomocí příkazu *set*. Hodnotami pro přiřazení mohou být cesty k souborům, textové řetězce či třeba pravdivostní hodnoty. Pokud není u proměnné uvedena nová hodnota pro přiřazení, pak je tato proměnná odstraněna. Určité systémové proměnné, jako je například *CXXFLAGS* v případě programovacího jazyka C++, uchovávají výchozí hodnoty – v tomto případě parametry pro překladač. Při prvním sestavení je obsah proměnné kopírován a uchováván v paměti jako *CMAKE_CXX_FLAGS*, proměnná *CXXFLAGS* je posléze ignorována *[25]*.

Příkazem pro nastavení možností překladu *add_compile_options* lze doplnit parametry platné na překládané soubory aktuálního a podřadných adresářů v pracovním prostoru. Zde zmíněné příznaky pro překlad jsou závislé na používaném typu překladače. Pomocí uvedených příznaků lze zajistit například vnímání upozornění při překladu jako chyby *[26]*.

Pro automatizované sestavení projektů v programovacím jazyce C/C++ je také možné využít nástroje Qbs. Význačný je opět využitím na více platformách, dále nezávislostí na vybraném programovacím jazyce a širokými možnostmi konfigurace navržené architektury. Konfigurační systém nástroje Qbs dovoluje návrh sestavení s vysokou mírou abstrakce, proto není nutné dbát na pořadí úkonů při sestavení, nýbrž spíše na jejich vzájemných vztazích. Sestavení se provádí dle obsahu konfiguračního souboru s koncovkou *qbs*, využívající dialekt QML. Zde se nachází členění do produktů (například konkrétní aplikace), které jsou součástí navrženého projektu. Typem produktu může být kupříkladu *CppApplication*, který signalizuje využití modulu *cpp*, ten je využíván pro návrh sestavení zdrojových kódů vytvářených v programovacím jazyce C++. Součásti tohoto modulu lze nastavit pomocí konvence *parametr: hodnota [27]*.

Výběr zdrojových souborů pro sestavení lze realizovat pomocí parametru *files*, předanou hodnotou parametru může být relativní cesta k souboru (ve vztahu k umístění konfiguračního souboru). Název souboru, včetně koncovky, je reprezentován textovým řetězcem, který je předán jako hodnota parametru *files*. Pro volbu verze standardu jazyka C++ lze v konfi-

guračním souboru definovat parametr *cxxLanguageVersion*, kterému lze předat hodnotu formou textového řetězce (např. tedy ‚*c++98*' nebo ‚*c++11*'). Pokud je tato vlastnost nastavena, pak budou při překladu přidruženy potřebné parametry vybraného standardu, jinak jsou využity výchozí hodnoty. Pro připojení dodatečných příznaků souvisejících s překladem projektu jazyka C nebo C++ lze využít parametr *cFlags*, který přijímá jednu anebo více hodnot, parametr *cFlags* je rovněž součástí modulu *cpp*. Kromě vyhledání a připojení souborů lze přidružit i dynamické knihovny. Parametrem *dynamicLibraries* modulu *cpp* lze připojit seznam dynamických knihoven, které je třeba při sestavení identifikovat. Pokud je však knihovna přímou součástí sestavovaného projektu, pak je doporučeno využít modul *Depends [28]*.

4.2 Kontrola paměti nástrojem Valgrind

Valgrind slouží jako analyzátor běhu programu, který zahrnuje sadu nástrojů pro kontrolu paměťového přístupu. Tyto nástroje pomáhají automaticky odhalit rozličné druhy paměťových chyb, mimo jiné i při zpracování pomocí více vláken, rovněž umožňují detailní profilování kontrolovaného programu. Valgrind zahrnuje nástroje pro kontrolu chyb paměti, větvení programu při práci více vláken či pomocí predikce, také pro profilování haldy (paměťového sektoru). Využití nástrojů Valgrind umožňuje docílení vyššího výkonu a omezení chybovosti navržených aplikací, nástroje jsou vhodné pro libovolný typ software bez ohledu na programovací jazyk (i když je Valgrind primárně zaměřen na jazyky C/C++). Nástroje je doporučeno využívat v rámci automatických testovacích sad, v případech provedení změn v programu či obecně při výskytu chyby *[29]*.

Nejvíce používaným nástrojem sady Valgrind je Memcheck, který slouží pro odhalení paměťových úniků. Spuštění paměťové kontroly lze provést pomocí příkazu *valgrind* v příkazové řádce před názvem spouštěného programu. Ačkoliv je nástroj Memcheck výchozí volbou z obsažených komponent, je možné jej explicitně vyvolat pomocí parametru *tool*. K chybám, které se k paměťové kontrole vážou, patří přístup do nepovolené části paměti, opouštění vymezených paměťových bloků, překročení vrcholu zásobníku, nebo například přístup do části paměti, která již byla uvolněna. Výsledek paměťové kontroly je uživateli zobrazen

pomocí textového rozhraní, pro výpis do souboru je možné k příkazu *valgrind* uvést parametr *log-file*. Hodnota ve formátu textového řetězce za tímto parametrem označuje cestu k souboru s výsledky paměťového testu.

Výstup paměťové kontroly nástrojem Memcheck uvádí, v případě nálezu, číselný identifikátor kontrolovaného procesu a informační řetězec popisující chybu. Po výpisu případných nálezů je obsaženo shrnutí počtu chyb dle klasifikace. Paměťové bloky rozpoznané jako únik jsou označeny jako stále dosažitelné (*Still reachable*) v případě, že je na daný blok po ukončení programu odkazováno. Tato situace (i když nemusí být vyloženě pokládána za chybu) předpokládá možné uvolnění paměti odkazovaného bloku. Případ zaručené ztráty paměťového bloku (*Definitely lost*) označuje neexistenci ukazatele na tuto část paměti, která proto nemohla být se skončením programu správně uvolněna. Situace nepřímé ztráty (*Indirectly lost*) označuje ztracený blok paměti, který nemůže být správně odstraněn důsledkem ztráty jiných částí paměti, jež na takový blok odkazují. Případ možné ztráty (*Possibly lost*) zahrnuje nález jednoho nebo více ukazatelů na blok paměti, přičemž alespoň jeden z nalezených ukazatelů je tzv. vnitřní (*interior-pointer*). Vnitřní ukazatel může na daný paměťový blok odkazovat pouze shodou náhod (nikoliv nutně cíleně), proto je takovou situaci třeba důsledně prověřit. Shrnutí nalezených chyb ve výstupu paměťové kontroly nástrojem Memcheck zahrnuje počet ztracených Bytů v každé ze čtyř kategorií paměťových úniků *[30]*.

4.3 Správa automatizované kontroly v systému GitLab

Systém GitLab představuje plně integrovanou vývojovou platformu pro týmovou tvorbu softwarových produktů. Zahrnuje sadu nástrojů nejen pro vývoj, ale také pro nasazení a průběžnou správu navržených aplikací. Poskytuje možnost uchování zdrojových souborů projektů se správou verzí, zpětnou vazbu vývojářům ve formě hlášení chyb či nových implementačních námětů, zobrazení vytvořených změn v rámci jednotlivých větví projektu, sestavování a testování prostřednictvím průběžné integrace (*continuous integration*) nebo například tvorbu statických webových stránek produktu. Navržené programové součásti jsou uchovány ve formě projektů, které lze nezávisle konfigurovat z hlediska viditelnosti a oprávnění uživatelů k úpravám zdrojových souborů. Rozsáhlejší projekty lze také sloučit do skupin, případně i podskupin *[31]*.

Průběžná integrace systému GitLab, označována jako CI/CD (*Continuous Integration/Continuous Deployment* resp. *Delivery*) je nástroj pro vkládání malých částí zdrojového kódu do obsahu zdrojových souborů projektu uchovaného v rámci GitLab a následné spuštění nastavené rutiny pro ověření správného sestavení nebo výsledků testů. Tohoto nástroje lze využít pro potvrzení očekávaného chování programu před integrací do hlavní větve programu. Konfigurovaná procedura se opakovaně spouští při každé změně souborů zahrnutých v projektu. Touto kontrolní praktikou lze zabránit vzniku neočekávaných chyb v rané fázi vývoje. Průběžnou integraci je možné konfigurovat zahrnutím souboru *gitlab-ci.yml* do kořenového adresáře projektu v systému. GitLab CI/CD umožňuje využití definovaných proměnných, plánování pravidelného spouštění kontroly programu, a především zobrazení či porovnání výstupních zpráv provedených kontrol (tzv. artefaktů) *[32]*.

Soubor *gitlab-ci.yml* slouží pro návrh automatické kontroly, definuje strukturu a pořadí vykonaných operací. Základní jednotkou jsou nezávisle prováděné úkony (*jobs*), neomezené z hlediska jejich počtu v konfiguračním souboru. Skupinové dělení úkonů zajišťují fáze kontroly (*stages*), jež jsou před začleněním úkonů pevně definovány. Úkony patřící do stejné fáze (označené klíčovým slovem *stage*) jsou zpracovány souběžně, každá následně definovaná fáze je zpracována až po úspěšném dokončení fáze předchozí – neúspěch jakékoliv fáze zabraňuje dalšímu vykonávání kontroly.

Pro práci se soubory v rámci kontrolní procedury je možné využít klíčového slova *cache*, které umožní lokalizaci souborů či adresářů pro dočasné uchování mezi zpracovávanými úkony. Cesty (*paths*) se uvádí z relativního hlediska v rámci umístění kontrolovaného projektu – vychází se z kořenového adresáře, kde se nachází soubor *gitlab-ci.yml*. Umístění definovaná mimo rozsah konkrétního úkonu jsou nastavena jako globální a poté přístupná všem úkonům v konfiguračním souboru. Funkci předání umístění nebo souborů ve výstupu kontrolní procedury zajišťuje klíčové slovo *artifacts*. Pokud je kromě cest (*paths*) definováno také klíčové slovo *when*, je možné rozlišit předání artefaktů (tj. souborů s výsledky) v případě úspěchu (*on_success*) či neúspěchu (*on_failure*) zpracovávaného úkonu, resp. je tak možné učinit vždy (*always*).

Obsah vykonávaného úkonu je určen klíčovým slovem *script*. Dále uvedený textový příkaz může tvořit jeden nebo více řádků, které je třeba v případě výskytu speciálních znaků (jako např. znaku dvojtečky) opatřit jednoduchými či dvojitými uvozovkami. Návratové hodnoty konfigurovaných skriptů je možné využít pro určení úspěšnosti úkonů. Pro vykonání příkazů

před nebo po zpracování operace klíčovým slovem *script* lze uplatnit definici *before_script*, resp. *after_script [33]*.

4.4 Alternativní metody kontroly správnosti kódu

Termín MOOC (z angl. Massive Open Online Course) označuje dosud rozporuplnou definici online výukových kurzů bez specifických vstupních požadavků, bez omezení z hlediska počtu účastníků, které zpravidla nejsou zpoplatněny. Zda se z těchto původních definic v praxi vždy vychází, závisí na interpretaci jednotlivých provozovatelů. MOOC ve svém smyslu, oproti nízkokapacitním online kurzům, předpokládá možný počet účastníků typicky v řádech jednotek až stovek tisíc, aniž by měla tato skutečnost vliv na funkcionalitu výuky. Význam otevřenosti online kurzů od počátku označoval dostupnost studijních materiálů účastníkům mimo univerzitu zodpovědnou za organizaci kurzu. Další pojetí otevřenosti vychází z konceptu otevřených vzdělávacích zdrojů OER (Open Educational Resources), z čehož vyplývá, že je obsah kurzů založen na bázi volně dostupných materiálů, proto je i samotný kurz zveřejněn skrze licenci s volnou dostupností, aby bylo možné obsah kurzů šířením dále využívat. Tento pojem otevřenosti je však rovněž často omezen konkrétními organizacemi, zejména jedná-li se o výdělečné výukové platformy. V takových případech mohou existovat bezplatné základní kurzy se zpoplatněným dodatkem, například ve formě certifikátu o splnění či individuální zpětné vazby od provozovatele kurzu. Klíčovou vlastností MOOC je především dálková účast v kurzu prostřednictvím internetového připojení, kdy není plnění předpokladů pro úspěšné dokončení kurzu řešeno kontaktní přítomností účastníků. S možností zápisu do kurzů z libovolné části světa přichází, jak vyplývá z názvu, masivní navýšení celkového počtu účastníků, které je ovšem doprovázeno vysokou mírou přerušení jejich absolvování *[34]*.

Codecademy je jednou z mnoha organizací, jež spadají do kategorie MOOC. Komunita je význačná zaměřením na tvorbu online kurzů návrhu software v různých programovacích jazycích s registrovaným počtem desítek milionů účastníků. V současnosti je v rámci projektu Codecademy Pro registrováno více než sto tisíc osob, účelem založení v roce 2016, více než čtyři roky po vzniku samotné Codecademy, byla především motivace k učení a rozvoji moderních programovacích dovedností s poskytnutím studijní osnovy pro naplnění cílů a potřeb členů, včetně interaktivních cvičení, kvízů, projektů a dalších prvků. Organizace se rovněž

zaměřuje na rozvoj učebních postřehů a tvorbu příležitostí pro spolupráci účastníků kurzů v rámci nabízených činností *[35]*.

Způsob kontroly správnosti zdrojového kódu v projektech účastníků, který se využívá v rámci Codecademy, zahrnuje využití systému GitHub. Především se jedná o připomínky k úryvkům kódu, které je třeba k studentské implementaci doplnit, naopak odstranit či jiným způsobem modifikovat. Pro kontrolu kódu se v rámci organizace také využívá tzv. ‚*pull requests*', tj. žádostí o přetažení větve (zkopírování provedených úprav) zdrojového kódu, které jsou poté kontrolovány jiným členem komunity. Nejčastěji je pro kontrolu projektů využito panelu chybového hlášení (GitHub issues). V profesionálním kontextu kurzů a v případě sdílených projektů je nástroje chybových hlášení využito pro lokalizaci defektů v software, případně pro návrh nových součástí implementace, kdy slouží jako seznam úloh k danému projektu. Osoby, které kontrolují obsah práce, proto zanechávají připomínky (a mnohdy přímé odkazy na řádky zdrojového kódu) účastníkům pomocí chybových hlášení systému GitHub, přičemž úložiště pro odevzdání zdrojových souborů projektů je členům kurzu před kontrolou organizací Codecademy dopředně vytvářeno *[36]*.

Sekce chybových hlášení v systému GitHub má obdobné vlastnosti jako schránka e-mailových zpráv s tím, že je možné poznámky sdílet a řešit skupinově v týmu. Detail chybové hlášky je charakteristický svým názvem a popiskem, hlášky lze označit barevnými štítky pro jejich kategorizaci a třídění. Rovněž je možné vytvářet milníky pro určení mezních termínů dokončení plánovaných činností, případně definují obsah, který musí být dokončen před vydáním další verze programu. Pomocí komentářů může vyjádřit svůj názor každý, kdo má k projektu v daném umístění přístup. V rámci textového obsahu chybových hlášení lze také využít referencí na jiné členy nebo týmy systému GitHub, čímž lze více obdobných problémů vzájemně sloučit. Informace o aktuálním stavu zpracování lze získávat prostřednictvím upozornění přes rozhraní systému GitHub nebo na uvedenou e-mailovou adresu *[37]*.

II. PRAKTICKÁ ČÁST

5 PŘEDMLUVA K VÝCHOZÍMU STAVU ÚLOH PROGRAMOVACÍHO JAZYKA C++

Úlohy v kurzu „Programování v jazyce C/C++“ pro studenty druhého ročníku FAI UTB ve Zlíně jsou v současné době strukturovány tak, že programování v C++ tvoří obsah poslední třetiny celkového trvání kurzu. Kurz je tvořen přednáškami se zaměřením na teoretické poznatky s ukázkami praktického řešení problematiky programování v jazycích C a C++, přičemž na cvičeních má student příležitost využít znalosti a procvičit jednotlivá témata, která tvoří náplň kurzu, v praxi. Pro tento účel jsou studentům připraveny ukázkové úlohy, které lze dále rozšířit v závislosti na zadání poskytovaném cvičícím. Studenti mají dále možnost vypracovat (dle zadání na cvičení) aktivitní úlohu, kterou poté prezentují cvičícímu a odevzdají do úložiště na webových stránkách výuky (LMS Moodle). Za splnění úkolů obdrží student bonusové body, které dále vylepšují jeho prospěch.

Úlohy programování v jazyce C++ byly před vypracováním této bakalářské práce pouze čtyři. Jelikož je C++ rozšířením jazyka C, jednalo se o úlohy spojené s novými vlastnostmi a principy objektově orientovaného programování. První úloha zahrnuje seznámení studenta s tvorbou třídy a jejími náležitostmi jako je konstruktor a destruktor, či členské proměnné a metody. Tyto principy si student procvičuje na ukázkové třídě zaměstnance, která zahrnuje jeho vlastnosti (např. plat) a metody pro zjištění těchto vlastností, nebo například pro výpočet celkové vydělané částky v době pracovního poměru. Druhá úloha implementuje třídu Student, která využívá kompozice a obsahuje třídu pro udržení data ve formátu dne, měsíce a roku. Tímto je možné uchovat při vytváření instance třídy Student například informaci o datu narození osoby, či o datu začátku studia. V této úloze si studenti na cvičení vyzkouší použití přátelských funkcí a přetěžování operátorů pro porovnání nebo odečítání dvou zadaných dat, například pro zjištění doby studia ve dnech. Třetí úloha slouží k procvičení dědičnosti v objektově orientovaném programování za pomoci virtuálních funkcí. U čtvrté a poslední úlohy je obsahem ukázka implementace ošetření výjimek v jazyce C++ (k výjimkám dochází například při neúspěšném pokusu o dynamickou alokaci paměti), ukázka využití šablon tříd a názorné využití knihovny STL (Standard Template Library) pro ukládání instancí třídy, úkol tohoto cvičení však není rozšířen o aktivitní úlohu.

Při zpracování těchto úloh si student vytváří vlastní nový projekt v doporučeném vývojovém prostředí a z webových stránek výuky kurzu vkládá úryvky kódu, které následně doplňuje pro zajištění očekávané funkcionality. Části kódu pro řešení jsou k dispozici v dokumentu

ve formátu PDF, které dále obsahují také užitečnou teorii k dané problematice. Při dokončení aktivitní úlohy provádí cvičící rozbor kódu studenta, načež uděluje bodové ohodnocení. V případě nedostatku času na dokončení zadání v průběhu cvičení lze úlohy prezentovat během následujícího týdne nebo zpětně před ukončením semestru.

5.1 Podnět k inovaci úloh jazyka C++

Kurz programování v jazyce C/C++ je určen převážně pro začátečníky či mírně pokročilé, tématika pokrývá syntaktické a obsahové základy u obou jazyků. Pro cvičení jsou vymezeny dvě hodiny týdně po dobu trvání semestru, což přibližně odpovídá třinácti týdnům. Z důvodu tematického rozsahu je sedm až osm týdnů věnováno programování v jazyce C, na samotný jazyk C++ zbývají čtyři až pět týdnů, je-li brán v úvahu i semestrální test.

Cílem rekonstrukce úloh v jazyce C++ je osamostatnění problematiky programovacího jazyka společně se zavedením nové kontrolní metody prostřednictvím jednotkových testů zaměřených na očekávanou funkcionalitu stávajících i nově vytvořených úloh, které dále rozšiřují povědomí studentů o možnostech programování v C++. V nově realizovaných úlohách studenti rovněž lépe uplatní využití šablon tříd, chytrých ukazatelů a práci s lineárním seznamem za pomoci knihovny STL. Studenti rozšíří řešení programu o algoritmy vyhledávání cesty pro pokrytí všech polí šachovnice při průchodu šachové figurky jezdce, v jiné úloze využijí kompozice tříd pro konstrukci součástí automobilu. Následně využijí externí knihovny SQLite pro zpracování příkazů jazyka SQL při údržbě lokální databáze, nebo v jazyce C++ implementují aplikaci hry života dle pravidel stejnojmenné hry využívající principy celulárního automatu.

Soubor jednotkových testů, který je součástí implementace každé úlohy, student získá se zadáním úkolu a základní kostry programu. Tu posléze rozšiřuje tak, aby splnil zadání a aby byly přiložené testy úspěšné. Testovací scénáře pokrývají kritické části programu a ověřují, zda bylo dosaženo požadovaných výsledků. Pro testování se využívá knihovny MinUnit.

Na rozdíl od nutné kontroly úlohy cvičícím je prvotní kontrola splnění zadání ověřována pomocí síťového repozitáře GitLab, kam student po dokončení práce svůj zdrojový kód nahrává. Nakonfigurovaná automatizovaná kontrola v řádech sekund ověří úspěšnost sestavení

projektu, splnění jednotkových testů a správné uvolnění paměti, k čemuž slouží nástroj Valgrind. Z tohoto síťového repozitáře si student rovněž získává šablonu úlohy, kterou dále pouze doplňuje – není proto nutné vytvářet nový projekt v rámci vývojového prostředí.

Inovované i nově vytvořené úlohy jsem zhotovil ve dvou verzích – studentská verze obsahuje základní strukturu programu se sadou jednotkových testů, které jsou v tomto stádiu standardně neúspěšné. Tuto verzi projektu má každý zaregistrovaný student k dispozici ke stažení v síťovém úložišti. Druhé provedení úlohy je ukázkové, obsahuje celkovou implementaci požadované funkcionality a zajistí úspěšný průchod testů. Cvičícímu může kompletní verze sloužit k porovnání se studentským zpracováním úlohy, studentům však není přístupna.

U rekonstruovaných i nově vytvořených úkolů studentské i kompletní verze je kladen důraz na udržení konzistentní úpravy kódu pro zajištění přehlednosti a pro motivaci studentů k zápisu dle ucelené struktury při programování. Kód je dle formální struktury dělen do příslušných částí (např. deklarace při tvorbě objektů umísťujeme do hlavičkových souborů), dále je například určena maximální délka řádku na 80 znaků a názvy členských třídních funkcí začínají názvem třídy. Pro formátování jsem použil nástroj Clang-Format, který lze integrovat do vybraných vývojových prostředí pro úpravu kódu v jazycích C, C++ nebo Objective-C.

Důležitým prvkem při tvorbě úloh pro studenty je rovněž poskytnutí dokumentace k částem zdrojového kódu, která kromě samotného návodu k řešení úlohy objasní probíranou problematiku. Dokumentace ve formátu generátoru Doxygen dělí zdrojový kód do více sekcí, zahrnuje doplnění hlavičky s popisem obsahu a jménem autora ke každému souboru se zdrojovým kódem a definuje například vstupní a výstupní parametry u členských metod v případě tříd.

U nových úloh není předpokládána vyšší náročnost při zpracování oproti dřívějšímu typu úkolů, samotná tématika totiž pokrývá syntaktické a funkční základy programovacího jazyka C++. Před absolvováním inovovaných úloh se ovšem, vzhledem k povaze jazyka C++, předpokládá dřívější zkušenost s programováním v jazyce C.

Nová struktura úkolů a kontrolní metodika vyžaduje před plným použitím ve výuce zkoušku ve vybraném okruhu studentů, která prověří robustnost zpracování úloh a celkovou spokojenost ze strany studentů i cvičícího. Z tohoto důvodu je součástí této bakalářské práce také

výzkum spojený s vytvořením statistiky úspěšnosti. Studenti na hodině cvičení absolvují vybraný úkol, výsledné řešení dle zadání poté nahrají do síťového repozitáře a porovnají své výsledky testů. Pomocí hromadné kontroly bude možné odhalit případné defekty v návrhu.

Návod pro zpracování každé z úloh je k nalezení v úložišti systému GitLab a rovněž na stránkách výuky kurzu, je tvořen podrobným popisem v českém a anglickém jazyce. Návod slouží k objasnění již implementovaných základních součástí kódu, které student dále rozšiřuje, a také k představení požadavků pro splnění úlohy. Součástí návodu mohou být také ukázky úryvků kódu, které studentům napoví v postupu tvorby jejich řešení.

Každý student má ve školním systému GitLab vlastní pracovní prostředí, odkud získá šablonu s výchozí podobou úlohy, po dokončení do stejného pracovního prostoru nahrává výsledek. Změna souborů zajistí kromě ověření sestavení také spuštění jednotkových a paměťových testů. Pomocí artefaktů, které kontrolní systém zanechává, může student zjistit umístění chyby bránící v sestavení, nebo které testy neprošly s úspěchem – na jejich základě může své řešení upravit.

6 VYPRACOVÁNÍ ÚLOH PRO KURZ PROGRAMOVÁNÍ V C++

Při vypracování nových verzí úloh, které vyplývají z existujících zadání, se využívá již definovaných očekáváných cílů, jež mají být studentskou implementací dosaženy. U nových úkolů, které nejsou podloženy existující šablonou, čerpám z odborné literatury, která slouží jako učební materiál obsahující vzorová řešení či náměty ke správným a ověřeným přístupům k programování v jazyce C++.

Ačkoliv jsou úlohy zadávány na cvičení pro zpracování v průběhu výuky, kdy cvičící objasňuje očekávanou funkcionalitu a může zodpovědět případné dotazy, lze je rovněž zadat jako téma domácí přípravy, kdy student využívá písemně formulovaných návodů či popisující dokumentace, která je součástí zdrojového kódu šablony projektu.

Úlohy inovované i nově vytvářené implementuji tak, aby jejich očekávaný výstup bylo možné ověřit pomocí jednotkových testů, které jsou součástí dalšího zpracování se zaměřením na poskytnutí co nejpoužitelnějších cvičebních materiálů.

6.1 Tvorba vzorových řešení a studentských verzí

Počátkem mé práce je specifikace a vytvoření struktury nových úloh. Jejich složitost musí narůstat rovnoměrně, u první úlohy předpokládám vzorové vypracování v průběhu cvičení. Mezi jednotlivými úlohami neexistují přímé vazby, dokončení předcházejících úkolů není podmínkou pro vypracování těch následujících, i když se dříve objasněné metody mohou vyskytovat ve více úlohách a je třeba předpokládat, že je student s těmito principy obeznámen. Obsah všech úloh vychází z praktik podložených odbornou literaturou, je co nejvíce obecný, aby bylo možné zmíněné přístupy aplikovat v různých situacích – nejen v konkrétních případech, které jsou využity v náplni úkolů.

6.1.1 Třída zaměstnance

Úloha konstrukce třídy zaměstnance (*Employee*) je první úlohou, která je studentovi poskytnuta. V rámci této úlohy (u které lze předpokládat společné vypracování za pomoci vyučujícího) je třeba implementovat prostou třídu s názvem *Employee*, která slouží k zobrazení základních vlastností osoby (pomyslného zaměstnance), v tomto případě věku, doby zaměstnání v jednotkách měsíců a jeho platu. Student si procvičí tvorbu třídy v jazyce C++, rozezná speciální členské funkce konstruktor a destruktor, dále využije přístupových funkcí k nastavení a získání členských vlastností. Kromě samotné implementace úlohy se student seznámí

se strukturou návodu k úlohám a stylem dokumentace dílčích částí zdrojového kódu. Tuto úlohu po dokončení vypracování potřebné funkcionality nahrává do systému GitLab a pozoruje způsob kontroly sestavení, výsledků jednotkových testů a rozboru paměti.

Vzorové řešení je tvořeno hlavním implementačním souborem *main.cpp* a soubory třídy *Employee*, jmenovitě tedy *employee.hpp* a *employee.cpp.* Tato podoba odpovídá struktuře dělení do souborů u původní úlohy i doporučeným praktikám, ze kterých čerpám.[1] V hlavičkovém souboru *employee.hpp* je obsažena deklarace třídy s dvěma typy konstruktoru – prázdným konstruktorem a parametrickým konstruktorem, který slouží k nastavení členských atributů přímo při vytváření instance třídy. Oba typy konstruktoru i destruktor vypisují při zavolání na standardní výstup textový řetězec, který umožňuje sledování posloupnosti volání těchto speciálních funkcí. Členskými atributy jsou tři číselné hodnoty datového typu *int* (z hlediska oboru viditelnosti nastaveny jako soukromé), které udržují informaci o věku, době zaměstnání a platu zaměstnance. Součástí deklarace jsou také (veřejné) přístupové členské funkce pro nastavení a čtení všech členských atributů. Rozšiřující členská funkce *Employee_GetTotalPayment* slouží k navrácení číselné hodnoty celkové platby, kterou zaměstnanec obdržel za dobu zaměstnání.

```
1   class Employee
2   {
3   public:
4       Employee();
5       Employee(int initAge, int initMonthsEmployed, int initWage);
6       ~Employee();
7
8       void Employee_SetAge(int age);
9       int Employee_GetAge();
10      void Employee_SetMonthsEmployed(int monthsEmployed);
11      int Employee_GetMonthsEmployed();
12      void Employee_SetWage(int wage);
13      int Employee_GetWage();
14      int Employee_GetTotalPayment();
15
16  private:
17      int mAge;
18      int mMonthsEmployed;
19      int mWage;
20  };
```

Zdrojový kód 1: Deklarace třídy zaměstnance

[1] Soubory jsou poté zahrnuty pomocí klíčového slova *include*, viz kapitola 1.1.1.

Definiční soubor *employee.cpp* zahrnuje implementaci deklarovaných součástí třídy. Parametrický konstruktor přiřazuje argumenty předané hodnoty členským atributům, stejnou činnost zajišťují nastavující členské funkce (pro každou členskou vlastnost odděleně). Definice členských funkcí pro získání nastavených hodnot věku, doby zaměstnání a platu je pro čtení z oblasti mimo jmenný prostor třídy nezbytná (vzhledem k nastavení členských vlastností na soukromé).[2] Definice členské funkce *Employee_GetTotalPayment* spočívá ve vynásobení doby zaměstnání v měsících a platu, tato číselná hodnota je poté navrácena.

```
#include "employee.hpp"

Employee::Employee() { cout << " -> Employee constructor is called." << endl; }
Employee::Employee(int initAge, int initMonthsEmployed, int initWage)
     : mAge(initAge), mMonthsEmployed(initMonthsEmployed), mWage(initWage) {
  cout << " -> Employee constructor (with values) is called." << endl;
}
Employee::~Employee() { cout << " -> Employee destructor is called." << endl; }

void Employee::Employee_SetAge(int age) { mAge = age; }
int Employee::Employee_GetAge() { return mAge; }
void Employee::Employee_SetMonthsEmployed(int monthsEmployed) {
  mMonthsEmployed = monthsEmployed;
}
int Employee::Employee_GetMonthsEmployed() { return mMonthsEmployed; }
void Employee::Employee_SetWage(int wage) { mWage = wage; }
int Employee::Employee_GetWage() { return mWage; }

int Employee::Employee_GetTotalPayment() {
  int totalPayment = mMonthsEmployed * mWage;
  return totalPayment;
}
```

Zdrojový kód 2: Doplnění definice třídy zaměstnance

V souboru *main.cpp* je vytvořeno jednoduché textové rozhraní, které se vypisuje na standardní výstup, s tvorbou instancí třídy *Employee*. V prvním případě je vytvořen objekt, kterému je pomocí přístupových funkcí pro nastavení členských vlastností přiřazena hodnota věku, doby zaměstnání a platu. Tyto číselné hodnoty jsou pomocí funkcí pro čtení vlastností vypsány na standardní výstup, obdobně je vypsána hodnota celkové vydělané částky. V případě druhé a třetí instance se využívá tvorby ukazatele na dynamicky alokované místo v paměti pomocí klíčového slova *new*, zde je využito parametrického konstruktoru s přímým přiřazením hodnot členských vlastností. Přestože tato metoda nese riziko opomenutí uvolnění alokované paměti klíčovým slovem *delete*, v úloze je využita jako ukázkový příklad možného řešení.[3]

[2] Z implementačního souboru *main.cpp* nelze k soukromým členům třídy přímo přistoupit, viz kapitola 1.2.2.
[3] Rozdíl obou principů spočívá mimo jiné v délce životnosti za běhu programu, více viz kapitola 1.2.4.

Úlohu zamýšlím jakožto prvotní seznámení studenta se základní syntaxí programovacího jazyka C++ a s vytvářením instance jednoduché třídy. Student ze systému GitLab získává šablonu této úlohy, která postrádá celkovou deklaraci třídy *Employee* v hlavičkovém souboru *employee.hpp*, stejně tak následnou definici ze souboru *employee.cpp* a zdrojový kód pro vytváření instancí ze souboru *main.cpp*. Součástí šablony projektu této úlohy je rozdělení do souborů, zahrnutí knihovny standardních vstupů a výstupů *iostream* pomocí direktivy *include*, a definice využití standardního jmenného prostoru direktivou *using*. Strohost součástí, které má student od začátku v šabloně k dispozici, je dána předpokladem společné implementace řešení na cvičení.

6.1.2 Přetěžování operátorů

Jedná se v předpokládaném pořadí zadávání o druhou úlohu, resp. první, kterou student vypracovává samostatně. Cílem je shrnutí problematiky skládání tříd, implementace členských funkcí pro přetížení operátorů a rovněž využití přátelské funkce. V úloze se také vyskytuje použití statické členské vlastnosti pro uchování počtu existujících instancí třídy *Student*. Třídu *Date* (datum), která je ve třídě *Student* zahrnuta, tvoří členské vlastnosti pro uchování číselné hodnoty pro den, měsíc a rok. Členské funkce třídy *Date* umožňují manipulaci se dvěma daty prostřednictvím přetížených operátorů. Spřátelená funkce zajistí přetížení operátoru ze zahrnuté knihovny *iostream*, která slouží pro předání členských vlastností třídy *Date* ve formátu řetězce pro další výpis v rámci třídy *Student*.

Projekt s úlohou je opět rozdělen do příslušných souborů pro oddělení obsažených komponent. Obsahem souboru *date.hpp* je zahrnutí knihovny *iostream* a definice užití jmenného prostoru *std* direktivou *using.* Je zde vložena deklarace třídy *Date* se soukromými členskými vlastnostmi datového typu *int* pro udržení informace o dni, měsíci a roku. Součástí je parametrický konstruktor, který přijímá číselné hodnoty dne, měsíce a roku pomocí vstupních argumentů a kopírovací konstruktor, který umožňuje nastavení členských vlastností nové instanci přímým přiřazením již existující instance v rámci hlavního programu. Členská funkce *Date_SetDate* je přístupovou funkcí, která vstupními argumenty nastaví členské vlastnosti třídy *Date*, funkce *Date_DisplayDate* nastavené hodnoty vypíše na standardní výstup. Deklarace členských funkcí s přetížením operátorů zahrnuje znaky rovnosti (==), ostré závorky (<, >) a mínus (-). Přátelská funkce vázaná na objekt *ostream* ze zahrnuté knihovny

iostream přetěžuje operátor dvojité ostré závorky (<<) pro formátování vypisovaných členských vlastností třídy *Date* do řetězce, který lze zobrazit standardním výstupem.[4] Toho se využívá při vytváření instance třídy *Student* a volání členské funkce pro výpis vlastností obsažené třídy *Date*.

V definičním souboru *date.cpp* jsou implementovány konstruktory (členské vlastnosti jsou nastaveny dle přijatých vstupních argumentů), členská funkce pro nastavení vlastností *Date_SetDate* a funkce *Date_DisplayDate* pro jejich výpis. Definice členské funkce přetížení operátoru rovnosti spočívá v porovnání hodnot dnů, měsíců a let volajícího objektu a (vstupním argumentem) přijatého objektu *Date* – v případě shody obou dat je navrácena booleovská pravdivostní hodnota *true.* Funkce přetížení operátoru ostré závorky (<) při porovnání těchto dat navrací pravdivostní hodnotu *true* v případě, je-li číselná hodnota data předaného vstupním argumentem nižší (pokud se jedná o datum narození, pak je osoba předaná argumentem funkce starší). Členská funkce druhé z ostrých závorek (>) vrací opačnou pravdivostní hodnotu první funkce. Funkce pro přetížení znaku mínus slouží pro návrat číselné hodnoty rozdílu dvou porovnávaných dat ve dnech – postup spočívá ve vzájemném odečtení počtu dní, měsíců a let. Implementace funkce má dvě řešení – jednodušší verze zanedbává rozdílnou délku měsíců (každý měsíc má tedy 30 dní) a neuvažuje ani přestupné roky. Přátelská funkce přetížení operátoru dvou ostrých závorek (<<) pomocí objektu *ostream* upravuje formát členských vlastností dne, měsíce a roku, aby je bylo možné tímto operátorem vypsat na standardní výstup.

```
1   bool Date::operator==(const Date& other) const{
2        if(mDay == other.mDay && mMonth == other.mMonth && mYear == other.mYear)
3             return true;
4        else
5             return false;
6   }
7
8   ostream& operator<<(ostream& output, const Date& date) {
9     output << date.mDay << "." << date.mMonth << "." << date.mYear;
10    return output;
11  }
```

Zdrojový kód 3: Definice funkce přetížení operátorů rovnosti a dvou ostrých závorek

Třída *Student*, která je deklarována v souboru *student.hpp*, obsahuje soukromé členské vlastnosti pro zobrazení jména studenta ve formě pole znaků s omezenou délkou a objekt *Date*,

[4] Využití přátelské funkce umožňuje přístup k privátním členským vlastnostem datumu, viz kapitola 1.2.2.

který uchovává informaci o narození studenta a o datu začátku jeho studia. Rovněž je deklarována statická číselná hodnota datového typu *int*, u které dochází k inkrementaci s každou novou instancí třídy *Student*.[5] Tato statická hodnota zůstává přístupná po celou dobu běhu programu. Třídu tvoří dva parametrické konstruktory, jeden pro jméno a datum narození, druhý rozšířen o datum začátku studia – číselné hodnoty jsou předány vstupním argumentem při vytváření instance třídy. K uvedeným deklaracím členských funkcí patří přístupové funkce *Student_SetStudyDate*, která nastavuje členskou vlastnost data počátku studia, *Student_GetBirthDate* a *Student_GetStudyDate* (pro navrácení objektu *Date* s hodnotami dat narození a počátku studia). Funkce *Student_DisplayStudyDate* a *Student_DisplayStudent* pak slouží pro výpis na standardní výstup. Statická členská funkce *Student_GetCount* navrací hodnotu počtu existujících instancí třídy.

V případě definice v souboru *student.cpp* jsou zpracovány parametrické konstruktory (přiřazením vstupních hodnot členským atributům), kde je rovněž inkrementována hodnota počtu existujících instancí třídy. Při volání destruktoru třídy je tato hodnota opět snížena. Definice funkce *Student_SetStudyDate* spočívá ve volání členské funkce třídy *Date*, která přebírá vstupní argumenty funkce *Student_SetStudyDate* a těmito hodnotami nastavuje členské vlastnosti pro den, měsíc a rok. Funkce *Student_DisplayStudent* vypíše informace o jméně a datu narození studenta na standardní výstup, *Student_DisplayStudyDate* pak jméno a datum počátku jeho studia. V rámci definičního souboru třídy *Student* je rovněž nastavena výchozí hodnota počtu studentů na 0, navrácení této hodnoty obslouží statická členská funkce *Student_GetCount.*

V hlavním implementačním souboru *main.cpp* je zahrnuta knihovna *ctime*, která poskytne přístup k aktuálnímu datu ze systému (toto datum je posléze využito pro porovnání s datem počátku studia a výpočtu, kolik dní od počátku studia dosud uplynulo). Vytváří se instance třídy *Date* dle hodnot zadaných parametrickým konstruktorem, ty jsou poté vypsány. Rovněž se vytváří instance třídy *Student*, u kterých je pomocí přetíženého operátoru ostrých závorek (<, >) dosaženo porovnání jejich věku. Dalším instancím třídy *Student* je prostřednictvím parametrického konstruktoru a přístupových funkcí přiřazeno datum počátku studia, které je pomocí přetíženého operátoru mínus (-) odečteno od aktuálního data, čímž je získán

[5] Je možné sledovat počet existujících instancí, aktuální hodnota je pro všechny instance shodná, viz 1.2.3.

dosavadní počet dní studia. V rámci vzorového řešení je dále vytváření pole ukazatelů na třídu *Student* (dynamická alokace), kdy je standardním vstupem postupně zadáváno jméno a datum narození studenta. Po ukončení tvorby nových instancí jsou informace o jménu a datu narození studentů vypsány na standardní výstup.

Utváření studentské verze úlohy spočívá v zachování deklarací tříd *Date* a *Student* v příslušných hlavičkových souborech *date.hpp* a *student.hpp*, zatímco v rámci definičního souboru *date.cpp* je zpracována implementace parametrického a kopírovacího konstruktoru třídy, destruktoru, přátelské funkce pro přetížení operátoru dvou ostrých závorek (<<), rovnosti (==) a ostrých závorek (<, >) pro porovnání dvou dat. Student implementuje přístupovou členskou funkci *Date_SetDate* pro přiřazení členských vlastností dle přijatých argumentů a dále vytváří definici členské funkce pro přetížení operátoru mínus (-). Implementace s ohledem na skutečný počet dní v měsících a přestupné roky je dobrovolná. Hlavní implementační soubor *main.cpp* je pro studenty upraven tak, že postrádá vytváření instancí třídy *Student* obsahující informace o datu počátku studia, přítomné nejsou ani operace závislé na přetížení operátoru mínus (-). Student také doplňuje dynamickou alokaci objektů třídy *Student* pomocí pole ukazatelů, rozšiřuje program o zadávání údajů o jménu a datu narození standardním vstupem a výstup poté zobrazuje.

6.1.3 Dědičnost a virtuální funkce

Třetí připravená úloha, která je zaměřena na uplatnění metod polymorfismu ve formě využívání dědičnosti tříd. Tato metodika je použita na příkladě tvorby pomyslné farmy zvířat. Každé zvíře je vyjádřeno třídou, které dědí vlastnosti a schopnosti svých předků. Ukázkou může být třída *Dog* (pes) a *Cat* (kočka), u kterých platí, že jsou zároveň savci a ještě obecněji, že jsou zvířata. Na základě skutečnosti, že jsou mnohé vlastnosti zvířat sdílené (resp. předané z rodiče na potomka), student rozšiřuje hierarchii vytvářením odvozených tříd. Při tvorbě je patrný vzájemný vztah mezi objekty – vztah „je“, uplatní se dědičnost a využívá se virtuálních funkcí.

Každá třída je opět strukturovaná deklaračním a definičním souborem, prvním stupněm hierarchie je třída *Animal*, představující nejvíce obecný objekt, slouží jako rodičovská třída svým specializacím. V souboru *animal.hpp* je obsažena deklarace třídy, která obsahuje členské vlastnosti popisující věk (*int*), hmotnost (*float*) a jméno (*string*). Z důvodu budoucího záměru vytvářet odvozené třídy jsou tyto vlastnosti nastaveny z hlediska viditelnosti jako chráněné (*protected*). Jsou natolik obecné, že je sdílí jakýkoliv případný potomek, který je

zvířetem.[6] Je obsažen prázdný konstruktor a destruktor, členské vlastnosti jsou nastavitelné přístupovými funkcemi *SetAge* (věk), *SetWeight* (hmotnost) a *SetName* (jméno). Členské funkce *GetAge*, *GetWeight* a *GetName* lze využít pro čtení dříve nastavených hodnot. Deklarována je rovněž členská funkce *Speak*, která poslouží pro vyjádření zvířecího popěvku – především však pro využití virtuálních funkcí, proto je označena klíčovým slovem *virtual*.[7] Obdobně je označen i destruktor. V definičním souboru třídy s názvem *animal.cpp* je provedena implementace konstruktoru (členským vlastnostem je přiřazena implicitní hodnota a volání speciální funkce je vypsáno na standardní výstup), destruktoru (kde je vypsán informační řetězec na standardní výstup) a členských funkcí. Přístupové členské funkce nastavují vlastnosti objektu dle přijatých vstupních argumentů, ty pro čtení je zase navracejí. Výjimkou je virtuální funkce *Speak*, které je definováno vypsání řetězce na standardní výstup.

Druhá vrstva hierarchie tříd je odvozenou třídou od *Animal*. Patří sem třídy *Mammal* a *Bird*, které představují savce a ptáky. Obě nové třídy dědí vlastnosti a funkcionality třídy *Animal*, třída *Mammal* ve svém deklaračním souboru *mammal.hpp* obsahuje dodatečný parametrický konstruktor, pomocí kterého lze při vytváření instance přímo nastavit věk zvířete. Destruktor i funkce *Speak* jsou opět označeny jako virtuální (z důvodu přehlednosti). Tato doplňující změna je u hlavičkového souboru třídy *Bird*, jmenovitě tedy *bird.hpp*, provedena analogicky. U definičních souborů *mammal.cpp* a *bird.cpp* je pak provedeno parametrické přiřazení hodnot členským vlastnostem a je zde také upraven tvar řetězců, aby místo třídy *Animal* reprezentoval třídy *Mammal* a *Bird* (např. popěvek je nově tedy ‚savčí' a ‚ptačí').

Třetí vrstva (nejvíce specializovaná) zahrnuje třídy, které dědí od tříd *Mammal* a *Bird* – tím pádem dědí i od základní třídy *Animal*. Patří sem příkladné druhy zvířat *Dog* (pes), *Cat* (kočka) a *Pig* (prase), které dědí od třídy *Mammal*. Od třídy *Bird* pak dědí zvířata *Chicken* (kuře) a *Duck* (kachna). Struktura těchto nových tříd je vzájemně obdobná – kupříkladu v hlavičkovém souboru třídy *Dog* (což je *dog.hpp*) je deklarace dodatečného konstruktoru pro tři parametry, které umožní nastavit věk, hmotnost a plemeno (plemeno psa je pro třídu *Dog* specifické, je utvořeno pomocí výčtového typu). Třídy jsou opatřeny virtuálním destruktorem. U třídy *Dog* je navíc možné plemeno psa nastavit a číst přístupovou funkcí *SetBreed* a *GetBreed*.

---

[6] Nižší stupně hierarchie tříd automaticky sdílí tyto vlastnosti a zahrnují i další, více viz kapitola 1.3.1.
[7] Označení provedeno pro zajištění pozdní vazby – správného výběru objektu, dle kapitoly 1.3.2.

```
1   enum DogBreed { DOG_DOBERMAN, DOG_TERRIER, DOG_MALAMUTE, DOG_BULLDOG };
2
3   class Dog : public Mammal {
4    public:
5     Dog();
6     Dog(int age);
7     Dog(int age, float weight, DogBreed breed);
8     virtual ~Dog();
9
10    int GetBreed() const;
11    void SetBreed(DogBreed);
12    void WagTail() const;
13    virtual void Speak() const;
14
15   protected:
16    DogBreed mBreed;
17  };
```

Zdrojový kód 4: Ukázka deklarace třídy *Dog*

Definiční soubor *dog.cpp* třídy *Dog* pak umožňuje implementaci celkem tří konstruktorů a přístupových funkcí nastavení a čtení plemene. Textové řetězce všech nových zvířat jsou opět patřičně předefinovány – například u třídy psa nyní volání funkce *Speak* vyniká štěkáním.

```
1   #include "dog.hpp"
2
3   Dog::Dog() : Mammal(), mBreed(DOG_DOBERMAN) {
4     cout << "  ---> Dog constructor." << endl;
5   }
6   Dog::Dog(int age) : Mammal(age), mBreed(DOG_DOBERMAN) {
7     cout << "  ---> Dog(int) constructor." << endl;
8   }
9   Dog::Dog(int age, float weight, DogBreed breed) : Mammal(age), mBreed(breed) {
10    mWeight = weight;
11    cout << "  ---> Dog(int, float, DogBreed) constructor." << endl;
12  }
13  Dog::~Dog() { cout << "  <--- Dog destructor." << endl; }
14
15  int Dog::GetBreed() const { return mBreed; }
16  void Dog::SetBreed(DogBreed breed) { mBreed = breed; }
17  void Dog::WagTail() const { cout << "Tail wagging..." << endl; }
18  void Dog::Speak() const { cout << "Woof woof." << endl; }
```

Zdrojový kód 5: Definice součástí třídy *Dog* v souboru *dog.cpp*

V souboru *main.cpp* jsou direktivou *include* vloženy hlavičkové soubory všech tříd. Nejdříve je vytvořena instance třídy *Dog*, nad kterou je volána virtuální funkce *Speak*. U instance třídy je dále pomocí přístupových funkcí vypsána hodnota věku a hmotnosti. Další dvě instance této třídy využívají pro přiřazení hodnot oba typy parametrických konstruktorů. Druhá část souboru zahrnuje tvorbu instancí dynamickou alokací paměti, přičemž ukazatele na objekty jsou ukládány pomocí datové struktury *vector* z knihovny STL (Standard Template Library). Po výběru zvířete prostřednictvím standardního vstupu je lze pojmenovat. Ukončení výběru tvorby nových zvířat je uzavřeno jejich výpisem a následným uvolněním paměti.

Studentům je projekt s úlohou poskytnut se shodným rozdělením do souborů jako v ukázkové verzi, které pro dokončení implementace doplňují. Studentská verze zahrnuje celkovou implementaci třídy *Mammal* (deklaračně i definičně), stejně tak třídy *Dog* a *Cat*. Cílem studenta je vytvořit třídu *Pig*, která rovněž dědí od třídy *Mammal*. Následně student implementuje třídu *Bird*, k ní vytváří odvozené třídy *Chicken* a *Duck*. Konečnou modifikaci představuje tvorba třídy *Animal* a nastavení této třídy jako rodičovské pro třídy *Mammal* a *Bird*. Implementační soubor *main.cpp* obsahuje předchozí ukázky, doplnit je třeba řešení s tvorbou instancí přes dynamickou alokaci a ukládáním do datové struktury *vector* s tím, že je využíváno ukazatele na třídu *Animal* namísto *Mammal*.

6.1.4 Šablony tříd a STL

Čtvrtou úlohu tvoří seznámení studenta s problematikou výjimek na demonstračním příkladě a využitím šablon tříd pro práci s různými datovými typy ve formě implementace struktury zásobníku, na jejímž základě student utváří šablonu pro hledání minima, maxima a výpočet součtu pole číselných hodnot různých typů. Dále student využívá vzorové implementace použití datové struktury dynamického pole (*vector*) v případě uchování ukazatelů na třídu *Animal* pro vlastní tvorbu ukládání za pomoci datové struktury seznamu (*list*), která je společně se strukturou dynamického pole obsažena v rámci standardní knihovny šablon STL. Ukázka programu obsahuje použití chytrého ukazatele *unique_ptr*, student využívá sdíleného chytrého ukazatele *shared_ptr.*

Ukázka využití výjimek zahrnuje třídu *myException*, která je odvozená od třídy *exception* a je deklarována v souboru *myException.hpp*. Deklarace je rozšířena o virtuální funkci *what*, která navrací pole znaků tvořící zprávu při vyvolání výjimky. Definice této členské funkce je obsažena v souboru *myException.cpp*.

Šablonu tříd zásobníku *Stack* tvoří deklarace v souboru *stack.hpp*, ve kterém je uvedeno klíčové slovo *template* pro užití objektu *T*, který slouží jako reprezentant libovolného číselného nebo vlastního definovaného typu, jež lze v rámci struktury zásobníku využít.[8] Tato třída obsahuje soukromé členské vlastnosti pro zobrazení vrcholku zásobníku a vlastní seznam

[8] Tato praktika zabraňuje redundantnímu zápisu třídy pro různé datové typy, viz kapitola 1.2.5.

položek, který je tvořen právě libovolným typem *T*. V deklaraci jsou uvedeny i veřejné členské funkce *Stack_Push* a *Stack_Pop*, které přijímají vstupní argument typu *T* a navracejí pravdivostní hodnotu na základě úspěšnosti operace, kterou funkce provádí. Pod deklarací třídy je uvedena explicitní tvorba instance pro každý datový typ, který je v rámci ukázky použit.

```
1   template <class T>
2   class Stack {
3     enum { maxsize = 100 };
4
5    public:
6     Stack();
7
8     bool Stack Push(const T&);
9     bool Stack_Pop(T&);
10
11   private:
12    int top;
13    T stackPtr[maxsize];
14  };
15
16  extern template class Stack<double>;
17  extern template class Stack<int>;
```

Zdrojový kód 6: Deklarace šablony třídy v hlavičkovém souboru *stack.hpp*

Definice zásobníku v souboru *stack.cpp* obsahuje vynulování členské vlastnosti vrcholku zásobníku v případě volání konstruktoru (tj. při tvorbě objektu). Členská funkce *Stack_Push* zapíše do pole typu *T* další položku, pokud již není překročen limit povoleného počtu existujících záznamů. Funkce *Stack_Pop* přiřadí položku na vrcholku zásobníku a sníží prvek ukazující na jeho vrchol, pokud je nějaká položka zapsána.

Třída *Animal*, která je využita pro ukládání do datové struktury dynamického pole a seznamu, je deklarována v souboru *animal.hpp*. Je tvořena prázdným a parametrickým konstruktorem, který přiřazuje předanou číselnou hodnotu členské vlastnosti pro uchování věku pomyslného zvířete. Třída disponuje i přístupovými členskými funkcemi *Animal_SetAge* a *Animal_GetAge*, které nastavují a získávají číselnou hodnotu (datového typu *int*) věku zvířete. Definice třídy je v souboru *animal.cpp*, oba konstruktory i destruktor při svém zavolání vypisují na standardní výstup textový řetězec, který indikuje vytvoření či odstranění instance třídy.

Šablona třídy *Calculator* (kalkulačka), která je předmětem práce studenta, obsahuje ve své deklaraci v souboru *array.hpp* opět určení objektu *T* pomocí klíčového slova *template*. Třídu *Calculator* tvoří soukromé členské vlastnosti velikosti pole s položkami (datového typu *size_t*) a pole objektů typu *T*. Konstruktor třídy je parametrický, jeho prvním argumentem je ukazatel na pole s prvky, druhým pak počet položek v tomto poli. Veřejné členské funkce

vrací prvek typu *T* s nejvyšší a nejnižší číselnou hodnotou, případně součet všech obsažených položek v poli. Další členská funkce zajišťuje výpis položek na standardní výstup. Obdobně jako u šablony tříd zásobníku, i u této deklarace je uvedeno explicitní vytvoření instance. V definičním souboru této třídy (*array.cpp*) je pak standardně uvedeno přiřazení vstupních argumentů konstruktoru členským vlastnostem třídy a tělo členských funkcí, které navracejí maximum, minimum a součet obsahu pole ve formě objektu *T*.

Deklarační soubor *list.hpp* neobsahuje vytváření třídy, jsou zde uvedeny deklarace funkcí, které jsou využívané pro uchování ukazatelů na třídu *Animal* pomocí datové struktury seznamu, kterou lze zahrnout z STL. Poté, co jsou direktivou *include* vloženy položky *list* a *memory* (pro využití chytrých ukazatelů), jsou zde uvedeny funkce *List_EnlistAnimals* (která zaplní seznam ukazateli na třídu *Animal* dle zadaného počtu položek), *List_AppendAnimal* (pro připojení ukazatele na konec seznamu), *List_PrependAnimal* (která novou položku umístí na začátek seznamu), *List_InvertAnimals* (pro otočení obou konců seznamu) a *List_DisplayAnimals* (funkce, která vypíše obsah seznamu na standardní výstup). Implementace v definičním souboru *list.cpp* využívá vestavěných členských funkcí třídy *list*, proto při zápisu na konec seznamu využívá funkce *push_back*, nebo například *push_front* při zápisu na začátek kontejneru. Pro průchod položkami v seznamu se využívá iterátoru, což je objekt, který ukazuje na konkrétní prvek v kontejneru, v případě funkce *List_DisplayAnimals* je iterátor posunován od začátku ke konci struktury postupnou inkrementací.[9]

V rámci souboru *main.cpp* se nachází využití jmenného prostoru *std* pro potřebné součásti, kterými jsou *vector, list, unique_ptr* a *shared_ptr.* V první části je implementována ukázka vyvolání výjimky, která je dána neúspěšným pokusem o alokaci velkého množství paměti. Dále je vytvořena instance šablony tříd *Stack* datového typu *double*, do kterého jsou vloženy a následně vypsány číselné hodnoty. Stejná třída je využita i pro datový typ *int*. Ve vzorovém řešení je dále obsažena tvorba instance třídy *Calculator* s vyhledáním maxima, minima a výpočtem součtu obsahu předaného pole datových typů *int*, *float* a *char*. V souboru *main.cpp* je v rámci ukázky dále uvedeno ukládání chytrých ukazatelů *unique_ptr* na třídu *Animal* pomocí datové struktury dynamického pole. Využití datové struktury seznamu je prováděno

[9] Posun iterátoru vpřed je prováděn znaky dvou plus (++), obdobně jako zvyšování hodnoty čísel, viz 1.5.2.

pomocí chytrých ukazatelů *shared_ptr*, které jsou platné do doby existence posledního ukazatele na daný objekt.[10]

Cílem studenta je implementace šablony tříd *Calculator*, k čemuž může využít již hotové šablony tříd zásobníku – z tohoto důvodu je kromě dokumentace popisující jednotlivé prvky deklaračního souboru *array.hpp* tento i definiční soubor prázdný, vyjma zahrnutí používaných knihoven či definice jmenného prostoru. Deklarace funkcí pro obsluhu datové struktury seznamu z knihovny Standard Template Library je ponechána jako u vzorového řešení, nicméně definiční soubor je prázdný – implementace těla funkcí zde zpracovává student. Soubor *main.cpp* obsahuje ukázku vyvolání výjimek, využití šablony tříd zásobníku a zaplnění datové struktury dynamického pole pomocí chytrého ukazatele *unique_ptr* na instance třídy *Animal* obdobně jako u vzorového řešení. Studentovi zde zbývá doplnit využití implementovaných součástí pro použití instance šablony třídy *Calculator* pro různé datové typy a kontejneru seznamu na základě definice funkcí, které jsou v souboru *list.cpp*, k čemuž uplatní chytrý ukazatel *shared_ptr*.

6.1.5 Jezdcova procházka

V úloze zabývající se obdobou problematiky Jezdcovy procházky rozšiřuje student návrh podoby šachovnice, která je tvořena jednotlivými poli, na kterých se může vyskytovat šachová figurka jezdce. Student implementuje mechanismus, pomocí kterého je možné s vybraným jezdcem v rámci šachovnice pohybovat, dle pravidel pouze pohyby odpovídající tvaru písmene „L“.[11] Student v této úloze využívá skládání tříd, ošetřuje podmínky provedení přesunu jezdce v závislosti na pozici v šachovnici a pravidlech a implementuje více verzí funkce pro navštívení všech polí v šachovnici. Implementaci jsem rozšířil o možnost vložení více jezdců, přičemž každé pole v šachovnici může být obsazeno maximálně jednou figurkou a příznak navštívení je poli udělen bez ohledu na to, kterým z jezdců bylo navštíveno.

Projekt je rozdělen na zdrojové soubory dle tvorby tříd. Na počátku je vytvořena deklarace třídy *Knight* (pro šachovou figurku jezdce), která je obsažena v hlavičkovém souboru

[10] Zde je také záměrně opomenuto uvolňování alokované paměti, neboť je automatické. Více viz kapitola 1.5.3.
[11] Přesun dále vyžaduje, aby bylo nové pole umístěno v šachovnici a nebylo dosud navštíveno, dle 1.4.1.

knight.hpp. Soubor začíná zahrnutím knihovny *iostream* direktivou *include* a následnou definicí jmenného prostoru *std*. Deklarace třídy *Knight* spočívá v uvedení soukromé členské vlastnosti typu *int* udržující číselnou hodnotu identifikátoru jezdce. K veřejným členským vlastnostem patří statické členy pro uchování počtu vytvořených instancí třídy a hodnoty naposledy přiřazeného identifikátoru jezdce. Třída *Knight* zahrnuje dva konstruktory, prázdný a parametrický (který při vytváření objektu jezdce přiřazuje hodnotu identifikátoru ze vstupního argumentu), rovněž výchozí destruktor. K členským funkcím třídy patří *Knight_GetIdInfo* a statické funkce *Knight_GetCount* a *Knight_GetLatestId*. V definici této třídy (v souboru *knight.cpp*) jsou statickým členským vlastnostem přiřazeny výchozí nulové hodnoty. Členská funkce *Knight_GetIdInfo* přijímá vstupním argumentem ukazatel na objekt této třídy a navrací hodnotu jeho identifikátoru. Statické členské funkce *Knight_GetCount* a *Knight_GetLatestId* umožňují čtení aktuálních hodnot počtu existujících instancí třídy a naposledy přiřazeného identifikátoru.

Další třídou, jejíž deklarace je v souboru *square.hpp*, je třída *Square*, která představuje jedno pole šachovnice. Na tomto poli může (ale nemusí) být umístěna figurka jezdce. Přítomnost figurky značí soukromá členská vlastnost typu *Knight*, která má v případě obsazení figurkou hodnotu jejího identifikátoru. Další soukromou členskou vlastností datového typu *int* je příznak navštívení tohoto pole. Je obsažen prázdný i parametrický konstruktor se dvěma vstupními argumenty pro přímé nastavení hodnot členských vlastností, rovněž je deklarován výchozí destruktor třídy. K veřejným členským funkcím patří *Square_GetIdInfo* a *Square_GetVisitedInfo*. Tyto funkce jsou v definičním souboru *square.cpp* implementovány pro příjem ukazatele na objekt *Square*, načež navrací hodnotu identifikátoru jezdce v tomto šachovém poli a informaci o navštívení tohoto pole.

Třída *Chessboard* sdružuje šachové pozice (*Square*) do matice. V deklaračním souboru *chessboard.hpp* jsou přítomny soukromé členské vlastnosti třídy dvourozměrného pole objektu *Square* a číselné hodnoty datového typu *int* pro uchování počtu řad a sloupců v šachovnici. Je deklarován parametrický konstruktor, který přijímá počet řádků a sloupců při vytváření šachovnice a destruktor. Členské funkce *Chessboard_AddKnight* a *Chessboard_RemoveKnight* slouží pro přidání a odběr figurky jezdce, přijímají souřadnice v šachovnici a navrací pravdivostní hodnotu dle úspěchu operace. Funkce *Chessboard_FindKnight* vyhledává jezdce v šachovnici, je deklarována pro příjem dvourozměrného pole datového typu *int* (reprezentující řádkovou a sloupcovou pozici v šachovnici) a identifikátoru jezdce. První z při-

jatých argumentů této funkce je i návratovou hodnotou. Členské funkce *Chessboard_MoveClockwise* a *Chessboard_MoveRandomly* slouží pro pravidelný a nahodilý pohyb jezdce v rámci šachovnice, vstupním argumentem těchto funkcí je identifikátor jezdce, návratem pak pravdivostní hodnota o úspěchu provedení přesunu. Funkce *Chessboard_MoveBacktrack* využívá při přesunu figurky jezdce dříve navštívených pozic (na které se v případě absence dalších platných přesunů figurka vrací), vstupním argumentem je identifikátor jezdce, návratovou hodnotou nejvyšší počet úspěšných přesunů figurky v šachovnici. Členská funkce *Chessboard_ShowChessboard* slouží pro grafické zobrazení šachovnice na standardní výstup.

```
#include "knight.hpp"
#include "square.hpp"

class Chessboard {
 public:
  Chessboard(int x, int y);
  ~Chessboard();

  bool Chessboard_AddKnight(int xPos, int yPos);
  bool Chessboard_RemoveKnight(int xPos, int yPos);
  int* Chessboard_FindKnight(int coordinates[2], int knightIndex);
  bool Chessboard_MoveClockwise(int knightIndex);
  bool Chessboard_MoveRandomly(int knightIndex);
  int Chessboard_MoveBacktrack(int knightIndex);
  void Chessboard_ShowChessboard() const;

 private:
  Square** mBoard;
  int mX;
  int mY;
};
```

Zdrojový kód 7: Deklarace třídy *Chessboard*

V definičním souboru *chessboard.cpp* jsou obsaženy implementace těla funkcí. Při volání konstruktoru třídy *Chessboard* je provedena dynamická alokace dvourozměrného pole šachových pozic (*Square*) pomocí operátoru *new* na základě přijatého počtu řad a sloupců. Posléze je každé z polí šachovnice obsaženo jezdcem se záporným identifikátorem (při návštěvě polí figurkou se pak tato záporná hodnota nahrazuje identifikátorem příslušného jezdce). Voláním destruktoru je uvolněna dynamicky alokovaná paměť (využívá se operátoru *delete*). Funkce *Chessboard_AddKnight* je definována ověřením, zda se přijatá pozice nachází v šachovnici a současně je prázdná, v případě platnosti je nový jezdec umístěn a statické hodnoty počtu existujících jezdců a posledního přiřazeného identifikátoru (které jsou součástí třídy *Knight*) jsou inkrementovány. Neúspěch operace je oznámen prostřednictvím výpisu textového řetězce na standardní výstup a navrácením pravdivostní hodnoty

false. U funkce *Chessboard_RemoveKnight* je opět ověřena zadaná pozice, v případě platnosti je poli nastaven záporný identifikátor jezdce a hodnota počtu existujících jezdců je snížena, neúspěch je signalizován výpisem textového řetězce a navrácením pravdivostní hodnoty *false*. Členská funkce *Chessboard_FindKnight* prochází matici šachových polí a navrací hodnotu řádku a sloupce takové pozice, která obsahuje identifikátor jezdce dle druhého vstupního argumentu. Při neúspěchu jsou navráceny souřadnice se zápornými hodnotami. Funkce *Chessboard_MoveClockwise* přesunuje figurku jezdce s argumentem přijatým identifikátorem dle vzoru písmene „L“ na první z maximálně osmi dostupných pozic, které jsou vybírány po směru hodinových ručiček od aktuální pozice figurky. Přesun na novou pozici může být proveden, když se nachází uvnitř šachovnice, tato pozice nebyla dosud navštívena a není obsazena jinou figurkou jezdce. Úspěšný přesun vede k navrácení pravdivostní hodnoty *true*, neúspěšný vrací *false* a oznamuje chybu výpisem textového řetězce na standardní výstup. Implementace *Chessboard_MoveRandomly* je od předchozí funkce odlišná nahodilým výběrem jedné z nanejvýš osmi možných pozic, jsou-li splněny podmínky pro validní přesun. Náhodný výběr pozice je možný zahrnutím knihovny *time.h* a následným využitím funkce *srand*, které se předává funkce *time* pro vytvoření časového razítka. Funkce *rand* je uplatněna pro výběr jedné z dostupných pozic pro přesun figurky. Úspěšný přesun jezdce navrací pravdivostní hodnotu *true*, neúspěch *false* – navíc je vypsán textový řetězec s chybou. U funkce *Chessboard_MoveBacktrack* je vstupním argumentem opět identifikátor jezdce, který má být přesunut. Součástí definice je dynamická alokace pro ukládání navštívených pozic (číselná hodnota řádku a sloupce) dané figurky. Při nalezení figurky s předaným identifikátorem pomocí funkce *Chessboard_FindKnight* je zaznamenána pozice do pole navštívených pozic a následně je proveden nahodilý pohyb s využitím funkce *Chessboard_MoveRandomly*. Je uchován počet úspěšných přesunů v řadě (z výchozí pozice), v případě dosažení konce možných přesunů před navštívením všech polí šachovnice se provádí návrat na předchozí pozici.[12] Úroveň hloubky navracení se odvíjí od počtu dosažení konců cesty, aniž by byla navštívena všechna pole – nanejvýš po startovní pozici figurky. Navrácena je číselná hodnota nejdelší nepřerušené cesty, i když nemusí odpovídat celkovému počtu polí v šachovnici. Funkce *Chessboard_ShowChessboard* je implementována průchodem dvourozměrného pole tvořící šachovnici s využitím textových řetězců pro výpis obsahu.

[12] Ukončení cesty před dosažením všech pozic je běžné například u šachovnice s 3×3 poli, viz kapitola 1.4.2.

Rovněž jsou vypsány hodnoty statických členských vlastností třídy *Knight* nesoucí informace o počtu existujících jezdců v rámci šachovnice a o posledním přiřazeném identifikátoru figurky.

V implementačním souboru *main.cpp* je vytvořeno jednoduché textové rozhraní pro obsluhu aplikace, přičemž jsou formou standardního vstupu zadávány číselné hodnoty, které reprezentují nabízené možnosti programu. Lze tak zvolit rozměry vytvářené šachovnice, ovládat vkládání či odstranění jezdců a následně vyvolat funkce pro pohyb figurky. Po vytvoření instance třídy *Chessboard* jsou v rámci hlavního programu volány členské funkce této třídy, kterým jsou předány vstupní argumenty (například tedy identifikátor jezdce) zadané uživatelem. Vstup je dále ošetřen pro příjem pouze číselných hodnot, které jsou v nabídce použity. Časová prodleva mezi jednotlivými kroky jezdce při jeho přesunu je zajištěna funkcí *usleep* z knihovny *unistd.h*, které je předán časový interval v mikrosekundách – tuto hodnotu lze rovněž za běhu programu změnit.

Cílem práce studenta je implementace členské funkce pro přidání jezdce na pozici v šachovnici, odstranění jezdce z šachovnice, nalezení a vrácení pozice jezdce dle zadaného identifikátoru a vypracování funkcí pro přesun jezdce výběrem pozice po směru hodinových ručiček a způsobem nahodilým. Bonusovým cílem je implementace funkce přesunu s navracením na dříve navštívené pozice při předčasném ukončení cesty. V závislosti na těchto předpokladech doplňuje student pouze definice těchto funkcí, proto upravuje pouze implementační soubor *chessboard.cpp*. Zbylé soubory jsou, včetně rozhraní v *main.cpp*, v porovnání se vzorovým vypracováním beze změny.

6.1.6 Kompozice a binární operace

Šestá vytvořená úloha slouží k procvičení skládání objektů do nadřazených celků. Pro tento účel existuje třída *Car*, které nese informace o automobilu, jež je tvořen doplňujícími prvky. Součástmi rozšiřující automobil jsou kola, křesla a motor. Součástí třídy *Car* je i členská vlastnost popisující příslušenství automobilu (volant, zrcátka, světlomety a rezervní kolo), které jsou uchovány datovým typem *unsigned int* a lze je nastavit pomocí binárních operací.

Třída s názvem *Wheel* reprezentující kolo má deklaraci v souboru *wheel.hpp*. Privátní členy této třídy nesou informace o počtu kol (*int*), jejich průměru (*float*) a šířce (*float*). Deklarovány jsou dva konstruktory – prázdný a parametrický, který umožňuje nastavení členských vlastností pomocí vstupních argumentů při vytváření instance objektu. Pro nastavení a čtení

vlastností jsou deklarovány přístupové členské funkce, které hodnoty pro nastavení přijímají ze vstupních argumentů nebo je navracejí jako výstup. Členská funkce *Wheel_ShowWheelInfo* vypisuje číselné hodnoty členských vlastností na standardní výstup. Součástí deklarace je přátelská funkce pro přetížení operátoru dvou ostrých závorek (<<) pro formátování textového řetězce vypisovaných členských vlastností při výpisu z nadřazené třídy *Car*. V definičním souboru *wheel.cpp* je implementováno přiřazení vstupních argumentů parametrického konstruktoru členským vlastnostem a obsah členských funkcí.

Třída *Chair* je deklarována v souboru *chair.hpp*. Reprezentuje sedadlo, které má soukromé členské vlastnosti popisující jejich počet (datovým typem *int*) a hmotnost (datovým typem *float*). Opět jsou uvedeny dva typy konstruktoru, výchozí a parametrický – pro přímé nastavení členských vlastností při tvorbě instance. Deklarovány jsou rovněž členské funkce pro jednotlivá přiřazení a čtení číselných hodnot počtu a hmotnosti sedadel. Funkce *Chair_ShowChairInfo* vypisuje hodnoty členských vlastností formou textového řetězce na standardní výstup. Obdobně jako u třídy *Wheel* je zde deklarována přátelská funkce pro přetížení operátoru dvou ostrých závorek (<<). V definičním souboru *chair.cpp* jsou pak implementovány členské funkce třídy *Chair* – vstupní argumenty parametrického konstruktoru jsou přiřazeny členským vlastnostem, je formulován textový řetězec pro výpis těchto vlastností funkcí *Chair_ShowChairInfo* a přístupové funkce modifikují jejich hodnotu nebo je navracejí svým výstupem.

Pro reprezentaci motoru je vytvořena třída *Engine*. V deklaračním souboru *engine.hpp* jsou uvedeny soukromé členské vlastnosti pro uchování informace o výkonu (*float*), obsahu motoru (*int*), maximální uváděné rychlosti (*int*) a spotřebě (*float*). Kromě prázdného konstruktoru je deklarován konstruktor parametrický pro příjem číselných hodnot pro nastavení členských vlastností. Vlastnosti lze nastavit nebo číst přístupovými členskými funkcemi, které jsou deklarovány pro každou vlastnost odděleně. I u motoru je deklarována funkce pro výpis členských vlastností na standardní výstup, která nese název *Engine_ShowEngineInfo*. V rámci třídy je deklarována přátelská funkce pro přetížení operátoru dvou ostrých závorek (<<) pro umožnění výpisu členských vlastností třídy *Engine* prostřednictvím třídy *Car*. Definiční soubor *engine.cpp* obsahuje implementaci přiřazení vstupních hodnot vlastnostem třídy jak parametrickým konstruktorem, tak přístupovými funkcemi. Tělo přístupové funkce pro čtení navrací číselnou hodnotu vybrané členské vlastnosti.

Třída *Car* sdružuje informace o vlastnostech automobilu a obsažených součástech. V deklaračním souboru *car.hpp* jsou uvedeny soukromé členské vlastnosti popisující model (*string*),

značku (*string*), rok výroby (*int*), počet najetých kilometrů (*float*) a příslušenství (*unsigned int*). Dále jsou v rámci třídy *Car* zahrnuty objekty *Wheel*, *Chair* a *Engine*.[13] V deklaraci je uveden výchozí a více parametrických konstruktorů (pro přímé nastavení vlastností třídy *Car* a dílčích součástí *Wheel*, *Chair* a *Engine*). Deklarovány jsou také členské funkce pro úpravu nebo čtení členských vlastností třídy *Car*, včetně funkce *Car_SetAccessories* pro nastavení hodnot pomocí binárních operací. Členská funkce *Car_ShowCarInfo* slouží pro výpis informací o automobilu i o součástech jako kola, sedadla a motor (jsou-li v automobilu zahrnuty), přičemž využívá přetíženého operátoru dvou ostrých závorek (<<), jež tyto součásti implementují. Vypsány jsou také položky příslušenství, které byly dříve přidány pomocí binárních operací. V deklaračním souboru třídy *Car* je také vytvořen výčtový typ pro přiřazení bitové hodnoty příslušenství automobilu. Definice třídy *Car* (v souboru *car.cpp*) rozšiřuje konstruktory o přidělení argumenty přijatých hodnot členským vlastnostem tříd *Car*, *Wheel*, *Chair* a *Engine* (vlastnostem, které nejsou v seznamu vstupních argumentů, se přiřazuje nulová hodnota). Nastavení, resp. navrácení hodnoty členských vlastností je implementováno i v přístupových funkcích. U funkce *Car_SetAccessories* je vstupním argumentem číselná hodnota datového typu *int*, která reprezentuje jeden ze čtyř dostupných doplňků do automobilu – po výběru doplňku vstupní číselnou hodnotou je členská vlastnost doplňků modifikována po jednotlivých bitech, vybrané příslušenství je potvrzeno výpisem informačního textového řetězce na standardní výstup.

V souboru *main.cpp* jsou ukázkově vytvořeny instance tříd *Wheel*, *Chair* a *Engine*, přičemž je pro nastavení členských vlastností využito jak parametrických konstruktorů těchto tříd, tak přístupových funkcí. Následně jsou vytvořeny instance třídy *Car* s využitím parametrických konstruktorů pro zahrnutí kol, sedadel nebo motoru. Implementována je rovněž tvorba instancí pomocí dynamické alokace paměti s možností výběru, které součásti budou do automobilu zahrnuty. Pro tento účel je vytvořeno jednoduché textové rozhraní, pomocí kterého uživatel (standardním vstupem) zadává informace o značce vozu, zahrnutí světlometů nebo například počtu kol.

Cílem studenta je samostatná tvorba součástí pro zahrnutí do automobilu včetně úpravy členské vlastnosti pro uchování informace o zahrnutém příslušenství vozu. Projekt se zadáním

[13] Třída *Car* využívá služeb tříd *Wheel*, *Chair* a *Engine*. Využívá se vztahu „má“ namísto „je“, viz 1.3.1.

úlohy obsahuje stejnou strukturu zdrojových souborů jako vzor. Deklarační soubor *wheel.hpp* třídy *Wheel* obsahuje uvedení přátelské funkce pro přetížení operátoru a deklaraci členské funkce *Wheel_ShowWheelInfo*. Definiční soubor *wheel.cpp* obsahuje pouze implementaci funkce pro přetížení operátoru a rovněž řešení funkce pro výpis *Wheel_ShowWheelInfo*. Obdobně jsou deklarační a definiční soubory upraveny pro třídy *Chair* a *Engine*. V deklaraci třídy *Car*, tj. v souboru *car.hpp*, je zahrnuto uvedení výčtového typu pro bitové přiřazení příslušenství automobilu. Jsou zde obsaženy členské vlastnosti a přístupové členské funkce, které nesouvisí s objekty kol, sedadel a motoru. Z konstruktorů je uveden prázdný a jeden parametrický, který nastavuje pouze členské vlastnosti značky, modelu, roku výroby a počtu najetých kilometrů.

```
#include "chair.hpp"
#include "engine.hpp"
#include "wheel.hpp"

enum Accessories {
  STEERING_WHEEL = 0x1,
  MIRRORS = 0x2,
  LIGHT_BULBS = 0x4,
  SPARE_WHEEL = 0x8,
};

class Car {
 public:
  Car();
  Car(string make, string model, int year, float driven);
  ~Car();

  void Car_SetMake(string make);
  string Car_GetMake();
  void Car_SetModel(string model);
  string Car_GetModel();
  void Car_SetYear(int year);
  int Car_GetYear();
  void Car_SetDriven(float driven);
  float Car_GetDriven();
  void Car_SetAccessories(int choice);
  unsigned int Car_GetAccessories();

  void Car_ShowCarInfo();

 private:
  string mMake;
  string mModel;
  int mYear;
  float mDriven;
  unsigned int mAccessories;
};
```

Zdrojový kód 8: Deklarace třídy *Car* (verze se zadáním úlohy)

Soubor *car.cpp* pak oproti vzoru nezahrnuje definici parametrických konstruktorů, které jsou závislé na dosud nevytvořených objektech kol, sedadel a motoru. Není obsažena ani definice funkce *Car_SetAccessories*, kterou implementuje student. V souboru *main.cpp* záměrně

chybí tvorba instancí tříd *Wheel*, *Chair* a *Engine*. Tyto dosud neimplementované součásti automobilu nejsou k dispozici ani při tvorbě instancí formou dynamické alokace paměti.

6.1.7 Databáze studentů

Tato úloha je vytvořena pro práci s externí knihovnou SQLite, která je zahrnuta jako součást projektu s úkolem. Cílem studenta je využití vestavěných funkcí knihovny (které jsou určeny pro programování v jazyce C/C++) pro ukládání vytvořených instancí třídy Student do trvale nebo dočasně uchovávaného souboru představující databázi. Trvale vytvořený soubor poté zůstává uložen v perzistentní paměti zařízení. Obsah souboru databáze je dále možné upravit přidáním nových záznamů, změnou vepsaných vlastností třídy Student nebo odstraněním záznamu. Funkce knihovny SQLite umožňují také výpis uložených součástí na standardní výstup za běhu programu.

Projekt s úlohou je opět rozdělen do souborů, které obsahují jednotlivé součásti zdrojového kódu. V hlavičkovém souboru *student.hpp* je obsažena deklarace třídy *Student*, tato třída představuje osobu se sadou vlastností. Soukromými členskými vlastnostmi, které jsou v tomto souboru deklarovány, jsou identifikátor (datového typu *int*), dále jméno (*string*), příjmení (*string*), ročník (*int*) a studijní průměr (*float*). Pro nastavení členských vlastností je deklarován parametrický konstruktor, hodnoty členů jsou předány vstupními argumenty. Pro čtení nastavených hodnot jsou deklarovány přístupové členské funkce. V definičním souboru *student.cpp* je implementováno přiřazení vstupních argumentů konstruktoru členským vlastnostem a jejich navrácení pomocí přístupových funkcí.

Pro využití externí knihovny je v kořenovém adresáři projektu obsažena složka *sqlite* se soubory *sqlite3ext.h*, *sqlite3.h* a *sqlite3.c*. Tyto soubory obsahují deklarační a definiční součásti knihovny SQLite, jejichž obsah není upravován a slouží pro zahrnutí do projektu s úlohou. Hlavičkový soubor *sqlite3.h* je společně s deklaračním souborem třídy *Student* (*student.hpp*) zahrnut pomocí direktivy *include* do deklaračního souboru třídy *Database*, kterým je *database.hpp*.[14]

Deklarace třídy *Database* zahrnuje soukromou členskou vlastnost typu *sqlite3* (struktura deklarovaná v souboru *sqlite3.h*) a číselnou hodnotu datového typu *int* pro uchování informace

[14] Není třeba provádět instalaci, stačí zahrnout uložené zdrojové soubory, více viz 1.5.4.

o otevření souboru databáze. Konstruktor třídy je parametrický, jeho vstupním argumentem je ukazatel na typ *sqlite3*. Třída obsahuje členské funkce využívající součásti knihovny SQLite. Deklarována je funkce *Database_OpenDatabase* pro otevření souboru navracející pravdivostní hodnotu dle úspěšnosti otevření souboru, jejím vstupním argumentem je řetězec znaků reprezentující cestu k souboru databáze. Další členskou funkcí je *Database_CreateStudentsTable* pro vytváření tabulky studentů, která navrací pravdivostní hodnotu. Pro vkládání záznamu do tabulky je deklarována funkce *Database_InsertStudent*, tato funkce přijímá ukazatel na objekt *Student* a navrací pravdivostní příznak dle úspěšnosti operace. Výpis zajišťuje členská funkce *Database_PrintTable*, která navrací pravdivostní hodnotu *false* v případě dosud neotevřené databáze. Funkce *Database_DeleteStudent* přijímá vstupním argumentem identifikátor studenta, jehož záznam posléze odstraňuje z otevřené databáze, výsledek operace je potvrzen navrácenou pravdivostní hodnotou. Pro úpravu existujícího záznamu v databázi je uvedena deklarace funkce *Database_UpdateStudent*, která přijímá identifikátor studenta, u kterého má být provedena změna, dále výběr položky pro změnu a novou hodnotu. Zavírání souboru databáze obsluhuje členská funkce s názvem *Database_CloseDatabase*, která rovněž navrací pravdivostní hodnotu dle výsledku operace.

```
#include "sqlite/sqlite3.h"
#include "student.hpp"

class Database {
 public:
  Database(sqlite3* database);
  ~Database();

  bool Database_OpenDatabase(char* path);
  bool Database_CreateStudentsTable();
  bool Database_InsertStudent(Student* student);
  bool Database_PrintTable();
  bool Database_DeleteStudent(int studentID);
  bool Database_UpdateStudent(int studentID, int changedItem, int intItem,
                              string textItem, float floatItem);
  bool Database_CloseDatabase();

 private:
  sqlite3* mDatabase;
  int mOpened;
};
```

Zdrojový kód 9: Návrh deklarace třídy *Database* pro práci s knihovnou SQLite

V definičním souboru *database.cpp* je implementován parametrický konstruktor, který přiřazuje vstupním argumentem přijatý ukazatel ke členské vlastnosti a nastavuje členskou vlastnost potvrzující otevření souboru databáze na 0. Členská funkce *Database_OpenDatabase* zahrnuje volání funkce *sqlite3_open* knihovny SQLite pro otevření databáze, v případě

úspěchu nastavuje členskou vlastnost stavu otevření souboru na hodnotu 1, chyba při otevírání je formulována textovým řetězcem, jež předává funkce *sqlite3_errmsg* knihovny SQLite.

```
1   bool Database::Database_OpenDatabase(char *path) {
2     int status;
3     if (mOpened == 0) {
4       status = sqlite3_open(path, &mDatabase);
5     } else {
6       cout << "Database is already opened." << endl;
7       return false;
8     }
9
10    if (status != SQLITE_OK) {
11      cout << "Can't open database: " << sqlite3_errmsg(mDatabase) << endl;
12      return false;
13    } else {
14      mOpened = 1;
15      return true;
16    }
17  }
```

Zdrojový kód 10: Definice členské funkce pro otevření souboru databáze

Definice členských funkcí pro vytváření nové tabulky studentů, vkládání nových záznamů, úpravu existujících záznamů, mazání záznamů a výpis obsahu tabulky využívá funkce SQLite *sqlite3_exec*, které je kromě typu *sqlite3*, označující aktuální databázi, předáno pole znaků nesoucí příkaz v jazyce SQL.[15] Funkce *Database_CloseDatabase* pro zavření databáze volá z knihovny zahrnutou funkci *sqlite3_close*, které je předán typ *sqlite3* dosud využívaného souboru pro ukládání instancí studentů. Součástí definičního souboru je také implementace funkce *toString* využívající šablonu pro definici typu *T*, který je předán jako náhrada číselných typů *int* a *float* při převodu vstupního argumentu funkce na formát textového řetězce. Pro tento účel je zahrnuta knihovna *sstream* a využito typu *ostringstream*. Funkce *toString* je využita při tvorbě příkazu SQL z přijatých číselných hodnot, které reprezentují vlastnosti studenta.

V souboru *main.cpp* je vytvořeno jednoduché textové rozhraní s výběrem operací nad bází dat, jednotlivé možnosti lze vybírat prostřednictvím standardního vstupu. Možnosti zahrnují otevírání souboru s ukládanými instancemi třídy Student, vytváření nové tabulky, výpis obsahu otevřeného souboru, vkládání nového záznamu do tabulky, úpravu již existujícího záznamu, smazání zvoleného záznamu z tabulky nebo uzavření souboru s ukončením běhu

[15] Pomocí funkce *sqlite3_exec* lze vykonat všechny transakční operace předáním řetězce, viz kapitola 1.5.4.

programu. Operace, které vyžadují otevření souboru s databází, nelze v případě uzavřeného souboru provést. V rámci hlavní smyčky programu je vytvořen ukazatel na typ *sqlite3*, ten je poté předán parametrickému konstruktoru při vytváření instance třídy *Database*. Instance třídy *Student* jsou vytvářeny dynamicky, pomocí operátoru *new*.

Úkolem zpracování je implementace členských funkcí třídy *Database*, které spravují obsluhu souboru s ukládanými instancemi třídy *Student*. Potřebné změny, které studenti na cvičení zpracují, se soustřeďují do souboru *database.cpp*. V porovnání s plnou verzí vypracování je v tomto souboru ponechána implementace funkce *toString*, je obsažena členská funkce pro otevírání souboru databáze, vytváření tabulky pro ukládání instancí třídy *Student* a funkce pro uzavření souboru databáze. Cílem studenta je proto implementace funkcí pro vkládání záznamů do tabulky *Student*, výpis obsahu tabulky na standardní výstup, mazání vloženého záznamu z databáze a úpravu existujícího záznamu v tabulce. Je třeba ukládat všechny členské vlastnosti třídy *Student*, tj. identifikátor, jméno, příjmení, ročník i průměr.

6.1.8 Hra života

V této úloze je využito skládání tříd pro implementaci obdoby hry života, tedy pro simulaci vývoje životního stavu buněk obsažených ve dvourozměrném poli. Tyto buňky, pro které je vytvořena třída *Cell*, jsou závislé na aktuálním stavu buněk ve svém okolí.[16] Množinu buněk pak reprezentuje třída *Matrix*, která seskupuje vytvořené buňky do dvourozměrného pole.

V deklaračním souboru *cell.hpp* je obsaženo zahrnutí knihovny *iostream* direktivou *include*, rovněž je deklarována třída *Cell* pro vyjádření stavu buňky. Jedinou členskou vlastností této třídy je soukromá booleovská pravdivostní hodnota popisující stav buňky, která může být živá (hodnota *true*) nebo mrtvá (hodnota *false*). Deklarační soubor obsahuje také uvedení bezparametrického a parametrického konstruktoru této třídy, parametrický vstupním argumentem přijímá pravdivostní hodnotu pro nastavení stavu buňky. Členská přístupová funkce *Cell_GetState* navrací životní stav buňky. V definičním souboru *cell.cpp* je v případě výchozího bezparametrického konstruktoru přiřazena členské vlastnosti životního stavu buňky hodnota *false* (buňka je tedy při vytváření instance mrtvá), u parametrického konstruktoru je členské vlastnosti stavu předán vstupní argument funkce.

[16] Za bezprostřední okolí se ve hře života pokládá tzv. Mooreovo okolí, tvořeno osmi sousedy, viz kapitola 1.6.

Soubor *matrix.hpp*, který je obsažen v projektu s úlohou, zahrnuje deklaraci třídy *Matrix*. V tomto souboru je direktivou *include* vložena knihovna *time.h* a *unistd.h* pro využití možností generování náhodné číselné hodnoty, resp. časové prodlevy za běhu programu. K soukromým členským vlastnostem třídy patří deklarace dvourozměrného pole objektu *Cell* a číselné hodnoty datového typu *int* pro udržení informace o počtu řádků a sloupců tohoto pole. Konstruktor třídy je parametrický, přijímá číselné hodnoty počtu řádků a sloupců pro vytvoření matice buněk, stejně tak údaj o jejich výchozím životním stavu. Členská funkce *Matrix_GetCellState* zajišťuje přístup k životnímu stavu buňky ve dvourozměrném poli, vstupními argumenty funkce jsou číselné hodnoty řádkové a sloupcové pozice. Pro změnu životního stavu buňky je deklarována funkce *Matrix_ChangeCellState*, která invertuje životní stav buňky na přijaté řádkové a sloupcové pozici. Pro změnu všech buněk v matici je deklarována funkce *Matrix_ChangeAllCells*. Členská funkce *Matrix_CountNeighbors* navrací číselnou hodnotu počtu živých sousedů vybrané buňky, přijímá řádkovou a sloupcovou pozici ve dvourozměrném poli buněk a číselný příznak pro zohlednění hranic matice. Buňky přesahující rozměry vytvořeného dvourozměrného pole lze uvažovat jako mrtvé (resp. neexistující), nebo lze počítat s jejich návazností na buňky z opačné strany matice – pak je generační vývoj buněk z hlediska pevných rozměrů plochy méně omezený.[17] Funkce *Matrix_CountAllCells* svým výstupem navrací počet všech živých buněk obsažených v matici. Pro generační vývoj buněk ve dvourozměrném poli je využito členské funkce *Matrix_EvolveCells*, která přijímá ukazatel na objekt *Matrix* pro uchování stavu předchozí generace, číselnou hodnotu očekávaného počtu generací, jež se mají provést, a příznak pro zohlednění hranic plochy. Funkce navrací počet provedených generací (v případě absence živých buněk jsou další předpokládané generace přeskočeny). Deklarovaná členská funkce *Matrix_ShowMatrix* zajišťuje výpis obsahu dvourozměrného pole s buňkami na standardní výstup.

```
1   #include "cell.hpp"
2
3   class Matrix {
4    public:
5     Matrix(int x, int y, int initialState);
6     ~Matrix();
7
8     bool Matrix_GetCellState(int row, int column);
9     bool Matrix_ChangeCellState(int row, int column);
10    void Matrix_ChangeAllCells();
11    int Matrix_CountNeighbors(int row, int column, bool borderless);
```

[17] Prvek, který buněčný prostor opustí, pokračuje ve své trajektorii protější stranou pole buněk, dle 1.6.

```
12      int Matrix_CountAllCells();
13      int Matrix_EvolveCells(Matrix* initialMatrix, int generations,
14                             bool borderless);
15      void Matrix_ShowMatrix();
16
17    private:
18      Cell** mCells;
19      int mX;
20      int mY;
21  };
```

Zdrojový kód 11: Deklarace třídy s maticí buněk v souboru *matrix.hpp*

Definiční soubor *matrix.cpp* rozšiřuje konstruktor třídy *Matrix* o dynamickou alokaci dvourozměrného pole buněk a nastavení výchozích pravdivostních hodnot jejich životního stavu. Všechny buňky mohou být při vytváření nové instance třídy nastaveny na živé či mrtvé, nebo může být jejich stav vygenerován náhodně. K tomu je využito zahrnuté knihovny *time.h*, kdy je funkcí *srand* vytvářeno časové razítko, pomocí kterého lze funkcí *rand* generovat posloupnost čísel 0 a 1 (jejich předáním parametrickému konstruktoru třídy *Cell* jsou čísla vyjádřena pravdivostní hodnotou *false* a *true*). Tím je životní stav každé z buněk v matici nahodilý. Dynamicky alokovaná paměť pro dvourozměrné pole je v destruktoru třídy uvolněna pomocí operátoru *delete*. Členské funkce využívající vstupních argumentů s číselnými hodnotami řádkové a sloupcové pozice navracejí v případě zadání souřadnic mimo hranice vytvořené plochy buněk pravdivostní hodnotu *false* nebo záporné číslo (v závislosti na datovém typu návratové hodnoty funkce). Všechny funkce rovněž volají členskou funkci třídy *Cell* s názvem *Cell_GetState* pro získání přístupu k členské vlastnosti životního stavu jednotlivých buněk. Princip implementace funkce generačního vývoje *Matrix_EvolveCells* spočívá ve tvorbě pomocné instance třídy *Matrix*, kam se ukládá podoba buněk pro novou generaci. Poté se kontroluje životní stav a počet živých sousedících buněk každého prvku v matici (s ohledem na volbu režimu hranic může jít i o sousedící buňky na opačné straně dvourozměrného pole), nový stav buňky je zaznamenán do pomocné matice dle pravidel hry života.[18] Na konci každé generace je pomocná matice nové podoby buněk ponechána jako aktuální, na jejímž základě se proces vývoje opakuje dle počtu očekávaných iterací. Přechod mezi jednotlivými generacemi je, kvůli čitelnému výpisu na standardní výstup zařízení, zpožděn pomocí funkce *usleep*, které je předán časový interval prodlevy v mikrosekundách. Funkce *Matrix_ShowMatrix* je utvořena pro výpis životního stavu buněk formou textového

[18] Pravidlem pro transformaci mezi generacemi je přechodová funkce *B3/S23* popsaná v kapitole 1.6.

řetězce, aby bylo možné pozorovat změny mezi jednotlivými generacemi. Vypsán je i aktuální počet živých buněk v matici.

V souboru *main.cpp* je vytvořeno textové rozhraní s ošetřením uživatelsky zadávaných číselných hodnot, které umožňují výběr z implementovaných funkcionalit programu. Vytváří se instance třídy *Matrix*, která je na základě zvolených rozměrů alokována dynamicky (pomocí operátoru *new*). V nabídce jsou možnosti tvorby nové buněčné plochy, zjištění životního stavu vybrané buňky, invertování stavu všech nebo vybrané buňky, zjištění počtu živých buněk, výpočet okolních živých buněk vybraného prvku a vývoj buněčného pole dle počtu požadovaných generací.

Úkolem studenta je implementovat členskou funkci třídy *Matrix* pro změnu životního stavu vybrané buňky dle předané pozice řádku a sloupce v buněčném poli, invertování stavu všech buněk, funkci pro získání počtu živých buněk a rovněž funkci pro zjištění počtu živých sousedících prvků vybrané buňky v obou režimech zohlednění hranic matice. Student zpracovává také členskou funkci pro generační vývoj buněk, opět pro oba režimy zohlednění hranic dvourozměrného pole. Pro tento účel postrádá verze projektu se zadáním úlohy implementaci členských funkcí *Matrix_ChangeCellState*, *Matrix_ChangeAllCells*, *Matrix_CountAllCells*, *Matrix_CountNeighbors* a *Matrix_EvolveCells* v souboru *matrix.cpp*. Funkce pro zjištění stavu vybrané buňky *Matrix_GetCellState*, zobrazení pomocí *Matrix_ShowMatrix*, konstruktor a destruktor třídy však zůstávají studentům k dispozici již dokončené. Soubor *main.cpp* s vytvořeným textovým rozhraním zůstává také v porovnání s verzí řešení úlohy nezměněn.

6.2 Tvorba testovacích scénářů a jednotkových testů

Za účelem zahrnutí testovacích souborů do projektů s ukázkovým řešením i zadáním pro studenty jsou kořenové adresáře úloh rozšířeny o podsložku *tests*, která obsahuje implementační soubor *tests.cpp*. V *tests.cpp* je, kromě hlavičkových souborů součástí pro testování, zahrnut také soubor *minunit.h*, který umožňuje využití zvoleného nástroje MinUnit.[19] Zmí-

[19] Pro využití nástroje MinUnit není nutná složitá instalace, postačí zahrnout hlavičkový soubor, viz 2.2.

něný soubor nástroje je rovněž obsažen ve složce *tests* u každého projektu. MinUnit lze využít pro porovnání výstupů členských funkcí testovaných tříd s očekávanými hodnotami, takže je možné ověřit, jsou-li implementované součásti v souladu se zadáním úlohy. Testovací soubor je u řešení i zadání úlohy shodný a pro studenty je viditelný. Všechny projekty s úlohami jsou dále konfigurovány tak, aby bylo po sestavení projektu možné nezávisle spouštět implementační soubory *main.cpp* i *tests.cpp* s výstupem jednotkových testů. Dle nástroje MinUnit jsou informace o výsledcích testů zobrazeny na standardní výstup formou znaku tečky v případě úspěchu, nebo jako chybová hláška při neúspěchu.

6.2.1 Návrh testovacích scénářů pro vytvořené úlohy

Testovací soubor *tests.cpp* každé úlohy je tvořen testovacími sadami, které se specializují na testování chování modulů při tvorbě instance vybrané třídy, případně jsou sady zaměřeny na konkrétní členské funkce tříd (v závislosti na rozsahu kontrolované třídy). Jednotlivé testovací případy, které jsou realizovány makrem definovanými funkcemi nástroje MinUnit (jako např. *mu_assert* nebo *mu_assert_double_eq*), vybírají očekávané výstupní hodnoty členských funkcí dle ekvivalentních tříd z celkové množiny výstupů, přičemž se uvažují i hraniční hodnoty.

U první úlohy, kde je cílem vytvořit třídu *Employee*, je v souboru *tests.cpp* zahrnut hlavičkový soubor *employee.hpp*. Obsaženy jsou dvě testovací sady, první (*employee_creation_testing*) zahrnuje testovací případ tvorby instance třídy *Employee* výchozím konstruktorem a následného přiřazení členských vlastností věku, doby zaměstnání v měsících a platu pomocí přístupových funkcí. Funkcí *mu_assert* nástroje MinUnit je ověřena implementace přístupových členských funkcí ve formě porovnání návratové hodnoty členské funkce pro čtení těchto vlastností s hodnotou předanou pro jejich nastavení.

```
1   MU_TEST(employee_creation_testing) {
2        Employee testEmployee;
3        testEmployee.Employee SetAge(25);
4        testEmployee.Employee_SetMonthsEmployed(14);
5        testEmployee.Employee_SetWage(25950);
6        mu_assert(testEmployee.Employee_GetAge() == 25,
7                  "Age was not set via function properly.\n");
8        mu_assert(testEmployee.Employee_GetMonthsEmployed() == 14,
9                  "Worktime in months was not set via function properly.\n");
10       mu_assert(testEmployee.Employee_GetWage() == 25950,
11                 "Monthly payment was not set via function properly.\n");
12    }
```

Zdrojový kód 12: Testovací případy k ověření implementace přístupových funkcí

V této sadě je dále vytvořena instance třídy *Employee* pomocí parametrického konstruktoru (s přímým nastavením členských vlastností), přiřazení předaných hodnot je opět testováno přístupovými členskými funkcemi, jejichž výstup je porovnáván s hodnotami uvedenými jako argumenty v konstruktoru. Druhá testovací sada s názvem *total_payment_testing* ověřuje funkcionalitu členské funkce pro výpočet celkové vydělané částky za dobu zaměstnání. Parametrickým konstruktorem třídy je vytvořena instance třídy, členská funkce *Employee_GetTotalPayment* je porovnána s násobkem vlastností počtu měsíců a platu, které byly předány konstruktoru. V případě společného vypracování úlohy na cvičení je obsah testovacích sad možné rozšířit o další případy navržené studentem.

Testovací soubor druhé úlohy obsahuje direktivou *include* vložené hlavičkové soubory *date.hpp* a *student.hpp*. První ze dvou testovacích sad, která nese název *studyDate_input_testing*, pomocí parametrického konstruktoru vytváří instanci třídy *Date*. Následně je parametrickým konstruktorem vytvořena instance třídy *Student*, hodnota členské vlastnosti pro datum studia je zvlášť předána přístupovou členskou funkcí. Poté je číselná hodnota nastaveného data počátku studia porovnána s výstupem přístupové funkce *Student_GetStudyDate*.

```
1   MU_TEST(studyDate_input_testing) {
2     Date studyDate(1, 9, 2015);
3     {
4       Student testStudent("Mike", 16, 2, 1994);
5       testStudent.Student_SetStudyDate(1, 9, 2015);
6       Date inputDate = testStudent.Student_GetStudyDate();
7       mu_assert(studyDate == inputDate,
8                 "Study date is not set via function properly.\n");
9     }
10  }
```

Zdrojový kód 13: Testovací sada pro ověření správné implementace přístupových funkcí

Druhá testovací sada *date_difference* ověřuje splnění zadání, které se týká přetížení operátoru mínus (-) pro výpočet rozdílu dvou dat. Každý případ vytváří parametrickým konstruktorem instanci třídy *Date*, které jsou od sebe odečteny pomocí přetíženého operátoru, výstup je porovnán s očekávanou hodnotou. Je zohledněna varianta pro uvážení 30 dní v každém měsíci i upřesněná metoda výpočtu rozdílu.

U úlohy *Inheritance_and_virtual_methods* jsou v testovacím souboru *tests.cpp* zahrnuty soubory *animal.hpp*, *bird.hpp*, *cat.hpp*, *chicken.hpp*, *dog.hpp*, *duck.hpp*, *mammal.hpp* i *pig.hpp*. Testovací sady jsou přizpůsobeny studentem implementovaným součástem a zaměřují se na jednotlivé třídy. Sada pro testování třídy *Pig* vytváří instanci výchozím konstruk-

torem a přístupovými metodami rodičovské třídy nastavuje členské vlastnosti věku a hmotnosti. Dva obsažené testovací případy ověřují předané číselné hodnoty s výstupem přístupových funkcí. Postup tvorby instance v oddělených testovacích sadách tříd *Bird*, *Duck*, *Chicken* a *Animal* je obdobný – instance vytvořená prázdným konstruktorem nastavuje členské vlastnosti věku a hmotnosti zvířete pomocí přístupové funkce, implementace přístupových metod je poté ověřena porovnáním nastavené hodnoty s výstupem přístupové funkce pro čtení vlastnosti. V sadě pro testování třídy *Animal* jsou dále utvářeny instance dynamickou alokací paměti prostřednictvím ukazatelů na odvozené objekty, kterým se také přístupovými funkcemi nastavují vlastnosti věku a hmotnosti a poté se kontrolují. Testovací sada *animal_name_test* vytváří instanci třídy *Animal*, které je přístupovou funkcí *SetName* nastavena členská vlastnost jména (datového typu *string*). Nastavená hodnota je porovnána s výstupem přístupové funkce *GetName*.

```
1  MU_TEST(animal_name_test) {
2    Animal testAnimal;
3    string name = "animalName";
4    testAnimal.SetName(name);
5    mu_assert(testAnimal.GetName().compare("animalName") == 0,
6              "Obtained name does not match. Name setting or obtaining was not "
7              "successful.\n");
8  }
```

Zdrojový kód 14: Testovací případ pro porovnání zadaného jména

U čtvrté vypracované úlohy, která obnáší mimo jiné využití šablon tříd, jsou v testovacím souboru *tests.cpp* zahrnuty hlavičkové soubory testovaných součástí *array.hpp* a *list.hpp*. První vytvořená testovací sada *class_template_tests* sdružuje testovací případy pro kontrolu funkcionality šablony tříd *Calculator*. Tyto případy v rámci přípravy před provedením testu definují pole datových typů *int*, *float* a *char* obsahující pevně dané hodnoty.[20] Následně se vytváří instance šablony tříd *Calculator* datového typu dle vytvořeného pole, které je vstupním argumentem předáno pole vybraného datového typu a počet jeho prvků. Pomocí porovnávací funkce *mu_assert* (a *mu_assert_double_eq* v případě pole s datovým typem *float*) je pro každý datový typ ověřeno nalezení největší hodnoty v poli, porovnán je výstup funkce *Array_GetMax* s očekávanou hodnotou. Obdobně je pro každý s datových typů *int*, *float* nebo *char* ověřeno nalezení minima, a to pomocí porovnání výstupu funkce *Array_GetMin*

[20] Jedná se o nastavení kontextu, který je nezbytný pro ověření vyvolaného chování, více v kapitole 2.1.1.

s hodnotou nejmenšího prvku v poli. U datových typů *int* a *float* je ověřena také funkcionalita členské funkce *Array_GetSum* pro navrácení součtu prvků v poli. V porovnávací funkci je předpokládána shoda výstupu *Array_GetSum* s očekávanou hodnotou.

```
{
  int intTestArray[8] = {14, -9, 2, 11, 74, 35, -52, 26};
  size_t itemCount = (sizeof(intTestArray) / sizeof(intTestArray[0]));
  Calculator<int> intCalculator(intTestArray, itemCount);
  mu_assert(intCalculator.Array_GetMax() == 74,
            "Maximum (int) was not obtained successfully.\n");
  mu_assert(intCalculator.Array_GetMin() == -52,
            "Minimum (int) was not obtained successfully.\n");
  mu_assert(intCalculator.Array_GetSum() == 101,
            "Sum (int) was not obtained successfully.\n");
}
{
  float floatTestArray[8] = {6.71f,  1.5f,    3.91f,   8,
                             69.71f, -26.71f, 37.115f, -17.55f};
  size_t itemCount = (sizeof(floatTestArray) / sizeof(floatTestArray[0]));
  Calculator<float> floatCalculator(floatTestArray, itemCount);
  mu_assert_double_eq(69.71, floatCalculator.Array_GetMax());
  mu_assert_double_eq(-26.71, floatCalculator.Array_GetMin());
  mu_assert_double_eq(82.685, floatCalculator.Array_GetSum());
}
```

Zdrojový kód 15: Úryvek testovací sady pro porovnání výstupu funkcí šablony tříd

Druhá testovací sada *list_tests* ověřuje zpracování funkcí pro využití seznamu s chytrým ukazatelem. Pro zjištění správnosti funkce *List_EnlistAnimals* jsou v testech použity hraniční hodnoty ekvivalentní třídy možných vstupů, proto se ověřuje možnost vložení nulového i záporného počtu položek, které správně nesmí projít.[21] Další testovací případ této sady ověřuje funkci *List_AppendAnimal* při přidání objektu *Animal* s hodnotou věku do seznamu, porovnávací funkce *mu_assert* ověřuje shodu očekávané hodnoty s výstupem přístupové funkce *Animal_GetAge*. Následně je ověřena funkce *List_PrependAnimal* při vkládání prvku na začátek seznamu, návratová hodnota funkce *Animal_GetAge* je opět porovnána s hodnotou, jež byla předána funkci *List_PrependAnimal*. Zjištění aktuální hodnoty je ověřeno i při otočení seznamu funkcí *List_InvertAnimals*.

Součástí souboru *tests.cpp* páté úlohy pro problematiku jezdcovy procházky je zahrnutí souboru *chessboard.hpp* pomocí direktivy *include*. Pro každou z členských funkcí třídy *Chessboard*, které jsou součástí zadání studenta, je implementována testovací sada obsahující podrobné testovací případy. Sada *knight_addition_tests* zahrnuje tvorbu instance třídy

[21] Hraniční hodnoty záporného i nulového počtu položek jsou v oddělených testovacích případech, dle 2.1.2.

Chessboard. Instance je využita pro vkládání figurek jezdců pomocí funkce *Chessboard_AddKnight* s předáním pozice řádku a sloupce v šachovnici. Výstup (pravdivostní hodnota na základě úspěšnosti operace) členské funkce je porovnán s očekávanou pravdivostní hodnotou pomocí makrem definované funkce *mu_assert* nástroje MinUnit. Takto je ověřeno chování funkce při vkládání platném, mimo hranice šachovnice nebo na pozici jiné figurky. Kontrolována je i hodnota statické členské vlastnosti počtu existujících figurek. V testovací sadě *knight_removal_tests* je opět vytvořena šachovnice, na jedno pole je vložen jezdec. Pomocí funkce *mu_assert* je v rámci několika případů ověřen výstup funkce *Chessboard_RemoveKnight* při předání řádkové a sloupcové pozice pro odstranění figurky, jak v případě existence na předané pozici, tak v situaci mimo hranice šachovnice.[22] Opět je kontrolována i hodnota statické členské vlastnosti pro uchování počtu existujících figurek.

```
1   MU_TEST(knight_removal_tests) {
2     Chessboard *chessboard = new Chessboard(6, 6);
3     Knight::mCount = 0;
4     chessboard->Chessboard_AddKnight(2, 3);
5     mu_assert(chessboard->Chessboard_RemoveKnight(2, 3) == true,
6               "A knight should be removed successfully.\n");
7     mu_assert(Knight::Knight_GetCount() == 0,
8               "Knight count should decrease after a deletion - was your knight "
9               "removed successfully?\n");
10    mu_assert(chessboard->Chessboard_RemoveKnight(8, 2) == false,
11              "Given positions (row) should not be accepted.\n");
12    mu assert(chessboard->Chessboard RemoveKnight(-3, 4) == false,
13              "Given positions (negative row) should not be accepted.\n");
14    mu_assert(chessboard->Chessboard_RemoveKnight(3, 12) == false,
15              "Given positions (column) should not be accepted.\n");
16    mu_assert(chessboard->Chessboard_RemoveKnight(4, -5) == false,
17              "Given positions (negative column) should not be accepted.\n");
18    mu assert(chessboard->Chessboard RemoveKnight(0, 1) == false,
19              "A non-existent knight should not be removed.\n");
20    delete chessboard;
21    chessboard = nullptr;
22  }
```

Zdrojový kód 16: Podoba testovací sady pro ověření úspěšnosti funkce mazání jezdce

Sada *knight_finding_tests* ověřuje navrácení číselné hodnoty řádku a sloupce pozice jezdce v šachovnici. Je vytvořena nová šachovnice, poté je funkcí *mu_assert* kontrolován výstup funkce *Chessboard_FindKnight* v případě přítomnosti i nepřítomnosti vyhledávané figurky. Sada *knight_clockwise_movement_tests* pro kontrolu pohybu figurky vytváří šachovnici, na vybranou pozici se poté vkládá figurka jezdce. Porovnávací funkcí je pak ověřen výstup funkce *Chessboard_MoveClockwise* s očekávanou pravdivostní hodnotou dle rozměrů ša-

[22] Testování platnosti i neplatnosti zajišťuje vyšší robustnost pokrytí testů a odhalí více defektů, viz 2.1.1.

chovnice a výchozí pozice jezdce. Ověřena je úspěšnost platného pohybu s následným zjištěním nových souřadnic figurky, kontrolováno je i chování funkce při nemožném přesunu figurky. V případě testovací sady *knight_random_movement_tests* je po vytvoření šachovnice a vložení jezdce kontrolována výstupní pravdivostní hodnota členské funkce *Chessboard_MoveRandomly*, opět jsou brány v potaz možné i nedostupné přesuny s následnou kontrolou pozice jezdce. Sada *knight_backtrack_movement_tests* slouží pro ověření funkcionality funkce *Chessboard_MoveBacktrack*, která je zadána jako bonusová. V rámci dvou testovacích případů je funkcí *mu_assert* porovnán nejvyšší počet přesunů dle vytvořené šachovnice s očekávanou hodnotou, kontrola probíhá i u příliš malé šachovnice, která neumožňuje jediný pohyb.

Úloha pro využití skládání tříd a binárních operací v souboru *tests.cpp* zahrnuje soubory *car.hpp*, *chair.hpp*, *engine.hpp* a *wheel.hpp* direktivou *include*. Testovací sada pro ověření ukládání informací o příslušenství automobilu po jednotlivých bitech s názvem *binary_operations_tests* pomocí parametrického konstruktoru vytváří instanci třídy *Car*, nad kterou je volána funkce *Car_SetAccessories*. Pomocí funkce *mu_assert* je poté ověřena shoda návratové hodnoty členské funkce *Car_GetAccessories* s očekávanou hodnotou pro každý typ příslušenství i pro jejich kombinace. Testovací sada pro kontrolu implementace součástí třídy *Wheel* je označena *wheel_composition_tests* a zahrnuje tvorbu instance. V rámci prvního případu jsou členské vlastnosti počtu, průměru a šířky kol nastaveny pomocí přístupových funkcí. Přístupovými funkcemi pro čtení jsou tyto hodnoty také porovnány s číselnými hodnotami, které byly předány při nastavování vlastností. Druhý případ, namísto přístupovými funkcemi, nastavuje vlastnosti parametrickým konstruktorem, které se poté opět kontrolují. Ve třetím případě testování třídy *Wheel* je použit kombinovaný parametrický konstruktor třídy *Car* obsahující objekt *Wheel*, přístupovými funkcemi pro čtení jsou vlastnosti počtu, průměru a šířky kol opět porovnány s předanými hodnotami. Testování nastavení členských vlastností tříd *Chair* a *Engine* je provedeno stejným způsobem, v oddělených případech po nastavení nejdříve přístupovými funkcemi, poté parametrickým konstruktorem objektu i nadřazené třídy *Car*. Poslední sada v testovacím souboru *united_composition_tests* ověřuje úspěšnost tvorby instance třídy *Car* prostřednictvím parametrického konstruktoru pro všechny obsažené objekty. Výstupní hodnoty přístupových funkcí pro čtení vlastností jsou funkcí *mu_assert* porovnány s očekávanou hodnotou, která byla konstruktorem nastavena.

Úloha databáze studentů má soubor *tests.cpp* rozšířen o zahrnutí hlavičkového souboru *database.hpp*. Testují se studentem zpracované členské funkce třídy *Database* pro vkládání

nových záznamů do databáze, čtení z databáze, mazání a úpravu záznamů. Testovací případy obsahují tvorbu nového souboru databáze a vytvoření instance třídy *Student*. Správnost vkládání záznamů probíhá testováním funkce *Database_InsertStudent*. První případ ověřuje úspěšnost vložení po otevření databáze pomocí *Database_OpenDatabase* a vytvoření tabulky funkcí *Database_CreateStudentsTable*.

```
1   MU_TEST(correct_table_insertion_test) {
2     sqlite3 *testDb = NULL;
3     Database testDatabase(testDb);
4     char databaseName[MAX_DB_NAME_LENGTH] = "firstTest.db";
5     Student *testStudent = new Student(11, "John", "Malloc", 3, 1.821f);
6     testDatabase.Database_OpenDatabase(databaseName);
7     testDatabase.Database_CreateStudentsTable();
8     mu_assert(testDatabase.Database_InsertStudent(testStudent) == true,
9               "A student should be inserted into database.\n");
10    testDatabase.Database_CloseDatabase();
11    delete testStudent;
12    remove(databaseName);
13  }
```

Zdrojový kód 17: Testování funkce vložení záznamu při platných podmínkách

Druhý případ kontroluje návratovou hodnotu testované funkce při opomenutí otevření souboru databáze, u třetího případu je vynechána tvorba tabulky, která je pro vkládání záznamu nutná. Další testovací sady pro ověření vkládání do tabulky porovnávají pravdivostní výstupní hodnotu funkce *Database_InsertStudent* při nedodržení požadovaného formátu předaného identifikátoru, jména, příjmení, ročníku a průměru studenta. Čtení záznamů tabulky členskou funkcí *Database_PrintTable* zahrnuje vytvoření instance třídy *Database* a porovnání výstupní hodnoty této funkce s očekávanou hodnotou (kromě platných podmínek) také v případě neotevřeného souboru databáze a neexistující tabulky v databázi. Testovací sady určené pro testování mazání záznamů z tabulky zahrnují tvorbu instance třídy *Database*, jednotlivé případy kontrolují návratovou hodnotu funkce *Database_DeleteStudent* s očekávanou hodnotou v situaci opomenutí otevření souboru databáze, nepřítomné tabulky v databázi a v případě očekávaného splnění funkce. Aktualizace záznamu studenta je testována porovnáním výstupní hodnoty členské funkce *Database_UpdateStudent* s předpokládanou pravdivostní hodnotou. Testovací případy zahrnují situaci neotevřeného souboru databáze a neexistující tabulky, je kontrolován i formát nově předaných vlastností identifikátoru, jména, příjmení, ročníku a průměru studenta.

Soubor *tests.cpp* u osmé úlohy pro zpracování hry života obsahuje kromě zahrnutí hlavičkového souboru nástroje MinUnit také *matrix.hpp*. Navržené testovací sady jsou zaměřeny na kontrolu implementace jednotlivých členských funkcí třídy *Matrix*. Při ověření funkcionality funkce *Matrix_ChangeCellState* jsou vytvořeny instance třídy s živými i mrtvými buňkami,

posléze je funkcí *mu_assert* nástroje MinUnit porovnán výstup členské funkce s očekávanou pravdivostní hodnotou (která je závislá na předané řádkové a sloupcové pozici ve dvourozměrném poli). Stav buňky, která byla změněna, je ověřován pomocí členské funkce *Matrix_GetCellState*, výstupní pravdivostní hodnota je opět porovnána s předpokládanou hodnotou. Sada pro testování funkce *Matrix_ChangeAllCells* obsahuje tvorbu instance třídy s neživými buňkami, načež je tato funkce zavolána. Po průchodu maticí se zjištěním stavu každé buňky je počet zaznamenaných živých buněk porovnán s očekávanou hodnotou, které odpovídá celkový počet buněk v matici. Opětovné zavolání funkce *Matrix_ChangeAllCells* předpokládá návrat hodnoty celkového počtu živých buněk na nulu. Pro otestování správného zjištění počtu živých buněk v okolí vybraného prvku jsou pro každý testovací případ vytvořeny instance třídy *Matrix* s nulovým nebo plným počtem živých buněk. Pro kontrolu jsou vybrány buňky v rámci matice včetně jejich okrajů, návratová hodnota funkce *Matrix_CountNeighbors* je porovnána s očekávanou hodnotou dle výběru prvku. K ověřování dochází i při zadání řádkových a sloupcových souřadnic mimo hranice buněčného pole, výpočet okolních prvků u buněk na okraji matice je zohledněn i v režimu návaznosti pole z protější strany. V testovacím souboru je dále kontrolována činnost funkce *Matrix_CountAllCells* – návratová hodnota členské funkce je porovnána s celkovým počtem buněk v matici v případě tvorby instance třídy s živými buňkami, u instance obsahující pouze neživé buňky pak s nulou.

```
1   MU_TEST(full_all_cells_count_test) {
2     Matrix testMatrix(6, 7, 1);
3     mu assert(testMatrix.Matrix CountAllCells() == 42,
4                 "Number of all living cells should equal 42.\n");
5   }
6
7   MU_TEST(empty_all_cells_count_test) {
8     Matrix testMatrix(25, 25, 0);
9     mu assert(testMatrix.Matrix CountAllCells() == 0,
10                "Number of all living cells should equal 0.\n");
11  }
```

Zdrojový kód 18: Testovací případy ověřující správnost funkce *Matrix_CountAllCells*

V případě tvorby instance s nahodilým rozložením živých a neživých buněk je před porovnáním výstupu funkce zpracován průchod dvourozměrným polem se zaznamenáním stavu každé buňky. Testování vývoje buněčného pole dle pravidel hry spočívá v porovnání návratové hodnoty počtu dokončených iterací funkce *Matrix_EvolveCells* s předpokladem dle vytvořené instance třídy *Matrix*. V rámci testu je do buněčného pole také vložen pohyblivý

vzor, kromě úspěšného dokončení definovaného počtu generací je funkcí *mu_assert* kontrolována cílová pozice tohoto vzoru. Ověření funkcionality je vypracováno i pro režim návaznosti buněčného pole z protější strany matice.

Pro přímou definici testů ve formě porovnání výstupů konkrétních členských funkcí testovaných tříd s očekávanými hodnotami bylo třeba ponechat tyto součásti deklarovány a upravit podobu definice, kterou zpracovává student. Zajištění funkcionality testů proto vyžaduje přesné určení rozdělení součástí do souborů (pro zahrnutí hlavičkových souborů do *tests.cpp*) a pevného určení názvu členů testovaných prvků. V případě některých úloh, kde zpracovává student celkový návrh třídy nebo vytváří třídy odvozené, deklarace testovaných součástí ve verzi se zadáním úlohy záměrně chybí. V takových úlohách je překlad programu zajištěn pomocí direktivy pro preprocesor *define*, která umožňuje dosud neimplementované součásti nezohledňovat. Testovací případy, které dříve neexistujících členských vlastností či funkcí využívají, mohou být po implementaci zpracované studentem direktivou *define* aktivovány.

6.2.2 Postup vypracování dle jednotkových testů

Navržené testovací sady mají prostřednictvím nástroje MinUnit sjednocenou strukturu a definovaný výstup, který má formát textového řetězce. Úspěšné vykonání testu je reprezentováno znakem tečky, zatímco selhání testu chybovou hláškou nebo vlastní zprávou, která se vypisuje na standardní výstup. Funkce *MU_REPORT* zobrazuje informace o počtu provedených přiřazení a stavu úspěšnosti.

U každé úlohy jsou testovány pouze součásti, které jsou určeny v jejím zadání. Pokud jsou studentem implementované součásti zpracovány dle zadání (kde odpovídají názvy, vstupní argumenty a návratové hodnoty funkcí apod.), není třeba testů využívat jako vzoru pro zpracování úlohy při snaze o jejich splnění. Kromě způsobu řešení dle zadání úlohy nebo vnitřní dokumentace kódu, která popisuje činnost členských funkcí, lze k vypracování přistoupit dle zajištění splnění jednotlivých testů.[23] Ty jsou totiž konstruovány tak, že jsou rozčleněny do testovacích sad dle testované jednotky, jsou řazeny sekvenčně dle doporučeného postupu

[23] Splnění testů navržených v souladu se zadáním (dokumentací) značí dosažení potřebné funkčnosti, viz 2.1.3.

vypracování a jsou vzájemně nezávislé. Student si může dále v průběhu vypracování navrhnout vlastní testovací případy pro zajištění vyšší robustnosti.

6.3 Tvorba dokumentace a formátování kódu

K nově vytvořeným úlohám programování v C++ jsou přidruženy návody s popisem již implementovaných součástí a požadavků na výslednou funkcionalitu, kterou zpracovává student. Pro objasnění činnosti jednotlivých částí zdrojového kódu všech souborů, které jsou součástí verzí s řešením i se zadáním úloh, jsou tyto soubory rozšířeny o dokumentaci odpovídající struktuře generované nástrojem Doxygen. Poskytnutá dokumentace může sloužit, kromě návodu, ke zpracování problematiky, kterou se úlohy zabývají. Zdrojové soubory všech úloh jsou dále upraveny tak, aby dodržovaly konzistentní formu zápisu kódu. Pro tento účel je využito nástroje Clang-Format (který může být rovněž integrován do vývojového prostředí).

6.3.1 Vytvoření návodu k úlohám

Návod je součástí projektů s řešením i zadáním úloh, vždy nese název *README.md* a umístěn je v kořenovém adresáři projektů s úkolem. Tento formát (*Markdown language*) nabízí možnosti úpravy částí textu v podobě změny velikosti písma, zvýraznění či oddělení do odstavců, kterých je využito při zobrazení návodu v systému GitLab. Návody jsou proto vytvořeny pro zvýraznění názvů úloh, podstatných a klíčových slov, nebo strukturování úryvků zdrojového kódu ze zadání úlohy do samostatných sekcí.

Obsah návodu je závislý na konkrétní úloze, první část však vždy obsahuje popis funkcionality, která je již v rámci úlohy zpracována. Po objasnění námětu úkolu je definováno zadání, které má student vypracovat. Zmíněny jsou například klíčové členské vlastnosti a funkce, které s dokončením úlohy souvisí, rovněž je od povinné práce rozlišena implementace dobrovolná. Návod některých úloh obsahuje i ukázku zdrojového kódu, který může sloužit jako šablona pro vypracování. Student je informován o nutnosti správného uvolnění dynamicky alokované paměti před odevzdáváním úlohy.

```
1   # Úkol 1 - Třída Employee #
2
3   Procvičte si základní syntaxi jazyka C++. Vytvořte třídu zaměstnanec (Employee)
4   s členskými proměnnými věk, doba zaměstnání a plat.
5
6   Členské proměnné udělejte soukromé (private) a vytvořte pro ně přístupové
7   metody. Vytvořte také konstruktor, který umožní inicializovat věk,
```

```
8   dobu zaměstnání a plat.
9
10  Dále vytvořte členskou funkci, která vrátí informaci, kolik si zaměstnanec
11  celkem vydělal za dobu zaměstnání. Vytvořte program, ve kterém
12  vytvoříte 2 zaměstnance, vypíšete jejich věk, plat, dobu zaměstnání
13  a jejich celkový příjem za dobu zaměstnání.
14
15  **Program musí se skončením vrátit všechnu alokovanou paměť **
16  **(nezapomeňte použít 'delete' v případě alokace pomocí operátoru 'new').**
```

Zdrojový kód 19: Úryvek souboru *README.md* tvořící návod k první úloze

Návody k úlohám jsou napsány v rámci jednoho souboru *README.md* jak v českém, tak v anglickém jazyce, aby bylo vypracování úlohy možné i v případě komunikace pouze světovým jazykem. Obsah návodu k úloze je v obou jazycích shodný, anglický popis je vytvořen i u případných ukázek zdrojového kódu.

6.3.2 Tvorba dokumentace zdrojového kódu

Deklarační i definiční soubory všech projektů s úlohami obsahují také popis všech vytvořených součástí ve formátu generátoru dokumentace Doxygen. Tento popis jednak člení soubory do příslušných sekcí, současně poskytuje stručné informace o implementované funkcionalitě kódu a dále rozšiřuje vytvořené návody k úlohám o podrobnější popis klíčových prvků. Veškerá dokumentace je vypracována v anglickém jazyce. Sekce souborů zahrnují záhlaví souborů, případné vkládání pomocí direktivy *include* a využití jmenných prostorů pomocí *using*, následně část pro deklarace, definice nebo vytváření instancí. Označení sekce ve zdrojovém kódu má podobu jednořádkového komentáře s dodatečným doplněním znaku pomlčky (-) po stanovenou maximální délku řádku.

```
1   /*Includes: ------------------------------------------------------------*/
2   #include <ctime>
3   #include <iostream>
4
5   #include "date.hpp"
6   #include "student.hpp"
7
8   /*Usage definitions: ----------------------------------------------------*/
9   using std::cin;
10  using std::cout;
```

Zdrojový kód 20: Ukázka členění souboru se zdrojovým kódem do sekcí

Dokumentace dílčích prvků zdrojového kódu má formu víceřádkového komentáře (bloku) programovacího jazyka C++ s doplňujícími značkami, které jsou pro formát Doxygen typické (za účelem rozlišení částí dokumentačních bloků).[24] Každý soubor obsahuje záhlaví s názvem souboru využívající značku *file*, jméno autora uvedené pomocí *author*, datum vytvoření souboru reprezentované značkou *date*, stručný popis předaný pomocí *brief* a následně značku *par* pro zmínku vlastnictví souboru.

```
/**
 * @file        date.hpp
 * @author      Tomas Bartosik
 * @date        19.6.2019
 * @brief       declaration of class Date
 * ************************************************************************
 * @par       COPYRIGHT NOTICE (c) 2019 TBU in Zlin. All rights reserved.
 */
```

Zdrojový kód 21: Hlavička s popisem deklaračního souboru *date.hpp* druhé úlohy

Dokumentace ve zdrojovém kódu je zásadně umístěna nad deklarací prvku, kterým může být uvedení deklarace třídy – uvádí se značka *class* s názvem třídy. V případě členů třídy, jmenovitě členských vlastností a funkcí, je základní použitou značkou *brief*, za kterou následuje stručný popis dané součásti. Členské funkce (včetně konstruktorů a destruktorů) mohou dále využít značku *param* pro popis vstupních argumentů funkce (pokud existují) obsahující zejména název, datový typ nebo účel použití. Pro každý vstupní argument funkce je vyhrazen jeden řádek v bloku dokumentace začínající právě značkou *param*. U funkcí s libovolnou návratovou hodnotou lze uplatnit značku *return*, za níž následuje textový řetězec tvořený datovým typem a popisem návratové hodnoty.

```
/*!
 * @brief Chessboard_AddKnight inserts new knight into chessboard,
 *          current count of knights and latest used identifier is increased
 *          in case of success
 * @param (int) xPos row index for insertion
 * @param (int) yPos column index for insertion
 * @return TRUE if a knight was added successfully, FALSE if requested
 *          position is occupied or out of chessboard borders
 */
bool Chessboard_AddKnight(int xPos, int yPos);
```

Zdrojový kód 22: Dokumentace nad deklarací členské funkce třídy *Chessboard*

[24] Víceřádkový blok s popisem je umístěn nad deklarací a využívá více značek (je podrobný), více viz 3.1.

6.3.3 Úprava formátu zdrojového kódu

Formátování zdrojového kódu je provedeno za účelem zvýšení čitelnosti a konzistence z hlediska stylu zápisu součástí úloh. Je využito ve vývojovém prostředí integrovaného nástroje Clang-Format s předdefinovaným stylem formátování Google. Dle struktury tohoto stylu je upraven zápis zdrojového kódu ve všech souborech projektů s úlohami. Odlišnosti mezi styly formátování spočívají například v zalamování dlouhých řádků, způsobu odsazení obsahu bloků kódu nebo umístění závorek.

Samotné části zdrojového kódu, které podléhají změně provedené formátováním, jsou zejména délky všech řádků. Ty jsou po úpravě nástrojem omezeny na maximální délku osmdesáti znaků, proto jsou delší textové řetězce automaticky zalamovány na následující řádek. Součástí stylu Google je dále odstranění nadbytečných prázdných řádků v souborech s kódem (mezi řádky je pak vždy maximálně jeden prázdný). Složené závorky označující začátek bloku kódu, na rozdíl od závorek ukončujících, nejsou umístěny samostatně na novém řádku. Zvolený styl současně zajišťuje úplné vynechání zápisu závorek v případě, je-li to možné – kupříkladu v situaci větvení programu podmínkou *if* obsahující jediný jednořádkový příkaz. Odsazení obsahu bloků, které jsou odděleny složenými závorkami, tvoří dvě mezery.[25] Formát zápisu zdrojového kódu u všech souborů shodným stylem zvyšuje přehlednost a zajišťuje lepší orientaci.

[25] Odsazení je vytvořeno mezerami namísto odsazení pomocí tabulátoru (o výchozí šířce čtyř mezer), viz 3.2.

7 SPRÁVA AUTOMATIZACE KONTROLY V SYSTÉMU GITLAB

Jako hlavní nástroj pro sestavení projektů při tvorbě úloh v programovacím jazyce C++ je využit CMake, pro tento účel jsou navrženy konfigurační soubory s názvem *CMakeLists.txt*. Kontrola úspěšnosti sestavení projektů je rovněž, pro každý úkol zvlášť, zajištěna pomocí konfiguračních souborů sestavení sady Qt Build Suite (využívající konfigurační soubory s příponou *qbs*). V rámci systému automatické kontroly lze předejít vykonání dalších úkonů v případě výskytu syntaktické chyby při tvorbě implementace úkolů studentem. Úkony, jež jsou kromě samotné kontroly úspěšnosti sestavení provedeny, dále zahrnují ověření výsledků jednotkových testů a rozbor paměti pomocí nástroje Valgrind. Konfigurace musí být přitom vytvořena tak, aby mohl student úlohy zpracovávat a spouštět nejen na platformě Windows, ale rovněž v systému Linux.

Prostor pro uložení vypracovaných verzí s řešením a zadáním inovovaných a nově vytvořených úloh je zajištěn v rámci systému GitLab. Pro každou vypracovanou úlohu jsou v tomto systému vytvořeny dva nezávislé projekty, jeden s kompletně vypracovaným řešením úlohy (který slouží jako vzor) a druhý postrádající implementaci požadované funkcionality. Obsahem každého projektu jsou, bez rozdílu mezi verzí s řešením a zadáním, podsložky *src* a *tests*. Podsložka s označením *src* obsahuje hlavičkové a implementační soubory v závislosti na konkrétní úloze, v podsložce *tests* je implementační soubor s jednotkovými testy a *minunit.h* (hlavičkový soubor nástroje MinUnit). Kromě podsložek jsou v projektech obsaženy konfigurační soubory Qt Build Suite (s koncovkou *qbs*, nesoucí název úlohy) a *CMakeLists.txt*. Pro následnou kontrolu úspěšnosti sestavení, jednotkových testů a rozboru paměti, je součástí projektů také konfigurační soubor *gitlab-ci.yml*. Dvojjazyčný návod k úlohám, ve formě souboru *README.md*, je v kořenovém adresáři projektu rovněž obsažen – znění návodu je automaticky zobrazeno po otevření stránky projektu s úkolem.

7.1 Konfigurace sestavení projektů

Výběr hlavičkových a implementačních souborů s dalším nastavením při sestavení projektu pomocí CMake je konfigurován v souboru *CMakeLists.txt*. V tomto konfiguračním souboru je vždy, klíčovým slovem *project*, uveden název úlohy. Deklarační a definiční soubory, které jsou umístěny v podsložce *src*, jsou v rámci konfiguračního souboru *CMakeLists.txt* do pro-

jektu zahrnuty pomocí příkazu *file*. Všechny v této složce obsažené zdrojové soubory s koncovkou *cpp* jsou pomocí klíčového slova *GLOB* zařazeny do seznamu *sources*. Cesty k hlavičkovým souborům s příponou *hpp* jsou obdobně klíčovým slovem *GLOB* sdruženy do seznamu, tj. do proměnné *headers*. Pro soubor *tests.cpp* (a případně další implementační soubory s jednotkovými testy) je vyhrazena proměnná s názvem *testSources*.[26]

```
1  project(composition_and_binary_operations)
2
3  file(GLOB sources "src/*.cpp")
4  file(GLOB headers "src/*.hpp")
5  list(REMOVE_ITEM sources ${CMAKE_CURRENT_SOURCE_DIR}/src/main.cpp)
6
7  file(GLOB testSources "tests/*.cpp")
```

Zdrojový kód 23: Výběr souborů při konfiguraci sestavení v *CMakeLists.txt*

Parametry pro překlad souborů se zdrojovým kódem jsou v rámci *CMakeLists.txt* předány příkazem *add_compile_options*. Zde jsou předány parametry *Wall* a *Wextra* pro zobrazení varovných hlášek, které jsou spojené s případnými chybami překladu, díky parametru *Werror* jsou tato upozornění vnímána jako chyby – překlad poté nemůže být úspěšně dokončen, nejsou-li případné chyby odstraněny. Pro využití standardu C++11 u úloh programování v jazyce C++ (které využívají náležité funkce jako například chytré ukazatele) je konfigurační soubor rozšířen o příkaz *add_definitions*, jež svým vstupním argumentem tento standard nastavuje. S tímto nastavením jsou také při sestavení vytvořeny spustitelné soubory hlavního implementačního souboru (*main.cpp*) a souboru jednotkových testů (*tests.cpp*).

U sedmé úlohy, tj. u tvorby databáze studentů, je konfigurace souboru *CMakeLists.txt* provedena s odlišnostmi. V adresáři *src* je obsažen další podadresář s názvem *sqlite*, kde jsou obsaženy deklarační a definiční soubory knihovny SQLite. Tyto soubory s příponami *c* a *h* jsou v rámci příkazu *file* také přidány do seznamu cest k souborům ukládaných v proměnných *sources*, resp. *headers*. Soubory knihovny SQLite obsahují i překladačem vypisovaná varování, proto v rámci příkazu *add_compile_options* není zahrnut parametr *Werror*, jelikož by nebylo možné projekt přeložit. Sestavení je kvůli parametrům pro překladač nastaveno odlišně pro systém Windows a Linux. V souboru *CMakeLists.txt* je obsaženo větvení podmínkou *if*, kdy jsou pro oba operační systémy přizpůsobeny parametry přidružené příkazem

[26] Znak hvězdičky (*), namísto názvu souboru, regulárními výrazy označuje všechny soubory, viz kapitola 4.1.

set. V rámci příkazu jsou k systémové proměnné *CMAKE_CXX_FLAGS* přiřazeny značky pro překladač dle operačního systému, na kterém se překlad provádí.

```
1  if(UNIX)
2      SET(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -pthread -Wl,--no-as-needed -ldl")
3  else()
4      SET(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -pthread -Wl,--no-as-needed -dl")
5  endif(UNIX)
6  add_compile_options(-Wall -Wextra)
```

Zdrojový kód 24: Větvení předávaných parametrů pro odlišné operační systémy

Použité značky dodávají podporu práce s více vlákny (*pthread*) a dynamického linkeru (*dl*, resp. *ldl*), které knihovna SQLite vyžaduje.[27]

Sestavení projektů s úlohami je v systému GitLab řešeno konfiguračním souborem sady Qt Build Suite využívající značkový jazyk QML. Soubory s nastavením sestavení projektů, které mimo jiné zahrnují výběr zdrojových souborů a předání parametrů pro překladač, nesou název úlohy s koncovkou *qbs*. Struktura souboru je u všech úloh obdobná – klíčovým slovem *Project* je vymezen obsah projektu, který obsahuje sekce *CppApplication* pro využití modulu *cpp*.[28] Dvě sekce (pojmenovány dle názvu úlohy a *tests*) oddělují, stejně jako konfigurace v souboru *CMakeLists.txt*, zdrojové soubory z podsložek *src* a *tests*. Cesty k souborům se zdrojovým kódem jsou uloženy do proměnné seznamu textových řetězců s názvem *sources*, kde je formou regulárních výrazů (znaku hvězdičky namísto názvu souboru) označen každý hlavičkový a implementační soubor s koncovkou *hpp*, resp. *cpp*. Obsah vytvořeného seznamu cest k souborům z podsložky *src* je přiřazen k parametru *files*, jež spadá do skupiny (*Group*) sekce *CppApplication*. U sekce jednotkových testů jsou tímto způsobem také vybrány všechny soubory, které jsou obsaženy v podsložce *tests*. Využití modulu *cpp* umožňuje přiřazení atributů pro překladač parametru *cFlags*, jež rovněž přijímá vytvořený seznam textových řetězců. Tento seznam (označený *flags*), stejně jako při konfiguraci pomocí CMake, obsahuje značky *Wall* a *Werror* pro výpis upozornění spojených s překladem a jejich vnímání ve formě chyb. K vlastnostem modulu *cpp* patří i parametr *cxxLanguageVersion*, kde je vyjádřena případná potřeba standardu C++11. Spustitelný soubor úlohy a

[27] Příkazem přidané značky jsou při prvním překladu sloučeny s hodnotou proměnné *CXX_FLAGS*, viz 4.1.
[28] Qt Build Suite podporuje více programovacích jazyků, proto modul *cpp* značí užití jazyka C++, více viz 4.1.

jednotkových testů je vytvořen dle takto definovaného nastavení, bez nutných změn konfigurace pro jednotlivé úkoly.

```
id: project
property stringList flags: ["-Wall", "-Werror"]
property stringList sources: ["src/*.cpp", "src/*.hpp"]
property string installDir: "bin"

CppApplication {
    consoleApplication: true
    name: "inheritance_and_virtual_methods"

    Group {
        files: project.sources
    }

    cpp.cFlags: project.flags

    Group {
        fileTagsFilter: "application"
        qbs.install: true
        qbs.installDir: project.installDir
    }

    Properties {
        condition: qbs.buildVariant == "debug"
        cpp.defines: "DEBUG"
        cpp.cxxLanguageVersion: "c++11"
    }
}
```

Zdrojový kód 25: Úryvek konfigurace sestavení projektu pomocí Qt Build Suite

Výjimku v konfiguraci tvoří opět úloha databáze studentů, kde jsou rovněž zahrnuty hlavičkové a implementační soubory patřící knihovně SQLite – ty jsou obsaženy v podsložce *sqlite* v rámci složky *src*. Soubory v adresáři *sqlite* jsou zahrnuty do seznamu textových řetězců s cestami k souborům s názvem *sources*, až poté jsou přiřazeny k parametru *files* sekce *CppApplication*. Ze seznamu argumentů pro překladač (*flags*) je u této úlohy odebrán argument *Werror*, obdobně jako v souboru *CMakeLists.txt* této úlohy. Další parametry pro překladač, které přidávají podporu vykonávání na více výpočetních vláknech a povolují dynamický linker, jsou předány parametru *dynamicLibraries* modulu *cpp*. Pro tento účel je vytvořena proměnná *flagDefines*, které je přiřazen atribut *pthread*. Využívá-li cílové zařízení operační systém Linux, pak je rovněž předán atribut *dl*.

7.2 Tvorba skriptů automatizované kontroly

Konfigurační soubory pro nastavení kontrolních procedur pomocí GitLab CI/CD jsou vždy obsaženy v kořenovém adresáři projektů s úlohami v systému GitLab. U všech vytvořených úloh, bez ohledu na verzi s řešením či zadáním úlohy, je podoba konfiguračního souboru

gitlab-ci.yml obdobná. Proces kontroly je tvořen úkony (*jobs*), které zahrnují ověření úspěšnosti sestavení projektu na obou platformách (tj. Windows i Linux), dále je obsaženo vyhodnocení výsledků jednotkových testů a rozbor paměti nástrojem Valgrind. Úkony probíhají v tomto sledu dle definovaných fází (*stages*). Pokud libovolná z těchto fází selhává, ty následující jsou automaticky přeskočeny. Na počátku konfiguračního souboru *gitlab-ci.yml* jsou definovány fáze (*stages*) s názvy *build* (kontrola sestavení), *unitTests* (pro vyhodnocení jednotkových testů) a *test* (rozbor paměti), fáze člení úkony do samostatných sekcí.

Fáze *build* zahrnuje kontrolu úspěšnosti sestavení v podobě provedení dvou úkonů: *linux* a *win*. V souboru *gitlab-ci.yml* je úkon *linux* přiřazen do fáze *build* pomocí klíčového slova *stage*. Využití klíčového slova *cache* umožňuje dočasné ukládání nezbytných součástí při sestavení projektu, klíčovým slovem *paths* je určena relativní cesta, v souvislosti s umístěním souboru *gitlab-ci.yml* v kořenovém adresáři úlohy, do adresáře úlohy (název podsložky odpovídá názvu úlohy) a jednotkových testů (zde je určen název *tests*). V rámci úkonu *linux* je také využito klíčového slova *artifacts*, kde jsou pomocí *paths* předány stejné cesty jako v případě *cache*. Uvedením tohoto klíčového slova dojde k předání výsledků úkonů mezi jednotlivými fázemi kontroly, kde s nimi mohou pracovat další definované úkony. Klíčovým slovem *when* je předán parametr *always*, takže výstupy jsou nahrávány bez ohledu na výsledek prováděného úkonu. V rámci obsahu klíčového slova *script* je v úkonu *linux* využito konfiguračního souboru pro sestavení pomocí Qt Build Suite s parametry, které potvrzují výběr sestavení odpovídající platformě Linux (využívá se sada překladačů GCC).

```
1   linux:
2     tags:
3       - seminars
4     stage: build
5
6     script:
7       - /opt/QtCreator/bin/qbs -f game_of_life.qbs profile:gcc
8       qbs.installRoot:. config:linux qbs.defaultBuildVariant:debug
9
10    cache:
11      paths:
12        - bin/game_of_life
13        - bin/tests
14
15    artifacts:
16      paths:
17        - bin/game_of_life
18        - bin/tests
19      when: always
```

Zdrojový kód 26: Část konfigurace automatické kontroly – sestavení v systému Linux

V případě úkonu *win* je opět uvedena příslušnost do fáze *build* klíčovým slovem *stage*.[29] V průběhu tohoto úkonu je zpracováno sestavení na platformě Windows, obsažené klíčové slovo *artifacts* definuje cestu (pomocí *paths*) ke spustitelným souborům s názvem úlohy a *tests*. Příkaz pro sestavení, opět následující za klíčovým slovem *script*, využívá konfiguračního souboru Qt Build Suite, předány jsou parametry pro provedení sestavení dle operačního systému Windows (je využito kompilátoru MinGW).

Kontrola výsledků jednotkových testů spočívá ve tvorbě úkonu s názvem *unitTesting*, který je klíčovým slovem *stage* zařazen do fáze *unitTests*. Vyhodnocení testů, za klíčovým slovem *script*, zahrnuje spuštění souboru *tests.cpp* a export výstupu do textového souboru *unitTestOutput.txt*. V tomto souboru je poté obsaženo shrnutí provedených jednotkových testů dle nástroje MinUnit. Úspěšnost je ověřena přítomností řetězce „*0 failures*“ v tomto souboru, který značí bezchybné provedení navržených testů. Textový soubor *unitTestOutput.txt* s výsledky jednotkových testů je klíčovým slovem *artifacts* zachován pro případné zobrazení, bez ohledu na úspěšnost kontroly.

```
1   unitTesting:
2     tags:
3       - seminars
4     stage: unitTests
5
6     script:
7       - bin/tests > unitTestOutput.txt
8       - if [[ `cat unitTestOutput.txt` =~ ", 0 failures" ]];then exit 0;
9       else exit 1;fi
10
11    artifacts:
12      paths:
13        - unitTestOutput.txt
14      when: always
```

Zdrojový kód 27: Část konfigurace automatické kontroly – zjištění výsledků testů

Pro paměťový rozbor nástrojem Valgrind je v souboru *gitlab-ci.yml* vytvořen další úkon s názvem *test1*, klíčovým slovem *stage* je přiřazen do fáze *test*. V bloku příkazů *script* je spuštěn příkaz *valgrind* s parametrem *log-file*, který výstup paměťové kontroly exportuje do textového souboru *valgrindOutput1.txt*.[30] Paměťová kontrola je vykonána u hlavní aplikace úlohy, kde student vytváří instance navržených součástí, případně u souboru s jednotkovými testy (*tests.cpp*). Dalším příkazem v sekci *script* je průchod souboru *valgrindOutput1.txt* a

[29] Náležitost úkonů *linux* a *win* do stejné fáze *build* umožňuje jejich souběžné vykonání, viz kapitola 4.3.
[30] Není-li u příkazu *valgrind* uveden parametr *tool* výběru nástroje, automaticky je vybrán Memcheck, viz 4.2.

vyhledání řetězce „*definitely lost: 0 bytes*“, který naznačuje, že nebylo dosaženo přímého paměťového úniku.[31] Textový soubor s výstupem je rovněž klíčovým slovem *artifacts* uchován v systému GitLab pro možnost detailního náhledu v případě potřeby, bez ohledu na úspěšnost paměťové kontroly.

```
test1:
  tags:
    - seminars
  stage: test

  script:
    - valgrind --log-file=valgrindOutput1.txt bin/tests
    - 'if [[ `cat valgrindOutput1.txt` =~ "definitely lost: 0 bytes" ]];then exit 0;
    else exit 1;fi'

  artifacts:
    paths:
      - valgrindOutput1.txt
    when: always
```

Zdrojový kód 28: Část konfigurace automatické kontroly – rozbor paměti

Vykonání automatické kontroly je, dle navrženého souboru *gitlab-ci.yml*, prováděno opakovaně při každé změně obsahu projektu s úlohou v systému GitLab. Vytvořením kopie projektu k automatickému spuštění kontroly nedochází, dokud nejsou obsažené soubory pozměněny, například tedy studentem v rámci vypracování vlastní implementace. Po vykonání kontrolních procedur lze z detailního náhledu projektu v systému GitLab přejít na podstránku kontrolních výsledků (*Pipelines*), kde lze sledovat stav úspěšnosti provedené kontroly a také stáhnout exportované soubory (artefakty) pro jejich případnou analýzu.

[31] Únik typu *Still reachable* se může vyskytnout u některých využitých knihoven, není proto testován, viz 4.2.

8 STATISTIKA ÚSPĚŠNOSTI VYPRACOVANÝCH PODKLADŮ

Po dokončení zpracování učebních materiálů pro výuku programovacího jazyka C++ bylo třeba implementaci vyzkoušet v reálné výuce, aby bylo možné posoudit kvalitu zpracování a objevit nedostatky, které lze odstranit. Pro tento účel jsem vybral jednu úlohu, na které si studenti, v rámci běžné výuky, měli vyzkoušet princip získání šablony se zadáním ze systému GitLab, následně úkol rozšířit o požadovanou funkcionalitu, prověřit své zpracování jednotkovými testy a pozorovat výsledky po zpětném nahrání do systému. Z osmi dostupných úloh jsem vybral první – třídu zaměstnance, právě z důvodu jednoduchosti (se záměrem společného vypracování na cvičení) a s možností pozvolného seznámení s objektovým přístupem programování. Vyzkoušení úlohy právě studenty, kteří kurz programování v jazyce C a C++ v tomto roce absolvovali, mělo ověřit přímočarost vytvořeného zadání a pokrytí jednotkových testů. Jakožto tvůrce materiálů jsem před vyzkoušením úkolu zamýšlel poskytnout studentům výklad spojený s potřebnou obsluhou systému GitLab pro získání a konečné vyhodnocení úlohy při odevzdávání. Po dokončení zpracování úlohy mají studenti vyplnit anonymní dotazník, pomocí kterého tak získávám zpětnou vazbu k nabídnuté úloze (s případnými náměty k vylepšení) a k novému způsobu zpracování obecně. Realizace byla zpočátku zamýšlena během probíhajícího kurzu „Programování v jazyce C/C++“, tj. v dubnu 2020, prezenční formou na učebně.

Z důvodu dočasného přerušení kontaktní výuky, které bylo spojeno s virovou epidemií COVID-19, nebylo možné úlohu zkoušet v rámci prezenční výuky kurzu „Programování v jazyce C/C++“ během letního semestru akademického roku 2019/2020. Studenti, kteří tento kurz absolvovali, proto obdrželi podklady pro zkušební vypracování úlohy distanční formou. Zadání úlohy tak mělo, namísto výkladu v průběhu hodiny cvičení, podobu podrobné prezentace ve formátu PDF, která obsahuje celkový postup vypracování (od získání šablony úkolu ze systému GitLab po výsledné odevzdání s pozorováním výsledků). Návod byl studentům zpřístupněn v profilu kurzu programování v univerzitním systému LMS Moodle, kde je obsažen také dotazník pro následnou analýzu úspěšnosti inovované úlohy a společné fórum pro případné diskuze spojené s komplikacemi při vypracování.

8.1 Tvorba materiálů pro distanční zkoušku úlohy

Předání zadání úlohy s tvorbou třídy zaměstnance předcházela tvorba studentských účtů v rámci využívaného systému GitLab. Z tohoto umístění studenti získali šablonu se zadáním úlohy (nikoliv s vypracovaným řešením), nacházeli zde vytvořený dvojjazyčný návod obsahující očekávanou výslednou funkcionalitu programu, do tohoto umístění své vypracované řešení také nahráli a poté sledovali úspěšnost zpracování. Studenti měli možnost neomezeného počtu výsledných odevzdání zdrojových souborů do systému, proto mohli své řešení upravovat až do fáze validního průchodu celé kontrolní procedury. K účtům ostatních studentů však neměl žádný student přístup, takže nemohl kopírovat práci svých kolegů. Připojení do využívaného studentského profilu systému GitLab je možné v případě připojení studenta do univerzitní sítě (například tedy v případě přítomnosti na fakultě), nebo prostřednictvím VPN (Virtual Private Network) z prostředí externí sítě.

Vytvořený návod k vypracování úlohy nahradil výklad s ukázkou řešení úlohy, který by proběhl v kontaktní výuce. V rámci tohoto dokumentu ve formátu PDF byli studenti seznámeni s potřebnými nástroji pro zpracování úlohy, s principem připojení a získání šablony se zadáním ze systému GitLab, dále se způsobem řešení úlohy při tvorbě třídy zaměstnance (včetně tvorby instance a ověření pomocí jednotkových testů) a se zpětným nahráním pro rozbor paměti a shrnutím výsledků. Kromě podrobného popisu bylo v návodu obsaženo velké množství snímků obrazovky pro sledování probírané problematiky. Návod popsal v ukázce využité nástroje (vývojové prostředí s překladačem jazyka C/C++ a CMake), rovněž konfiguraci připojení přes VPN (s odkazem na návod z externího zdroje). Studenti byli dále v návodu odkazováni na umístění projektu v systému GitLab, odkud zadání stahovali do svého zařízení. Následovala ukázka otevírání zadání v použitém vývojovém prostředí a seznámení se strukturou adresářů a zdrojových souborů projektu. Hlavní část prezentace znázornila samotnou implementaci třídy zaměstnance, zde byl studentům vysvětlen princip deklarování a následné definice součástí třídy a tvorba instance. Navržené součásti bylo pomocí jednotkových testů možné ověřit přímo v průběhu programování. Dle zadání v systému GitLab (a dokumentace ve formátu generátoru Doxygen v hlavičkovém souboru třídy) byli studenti nabádáni k samostatné implementaci členské funkce pro zjištění celkové výše platu za dobu zaměstnání, řešení bylo však posléze v prezentaci také vypracováno. Závěrem, po kladném vykonání všech jednotkových testů, popsal návod způsob zpětného nahrávání projektu s řešením do systému GitLab. Po nahrazení souborů ve studentském profilu s úlohou byla spuštěna automatická kontrola, která (v případě vypracování dle návodu) prokázala

úspěch všech ověřovaných součástí. Dílčí části výstupů kontroly byly také v prezentaci zobrazeny. Obsahem prezentace byla také žádost o vyplnění krátkého anonymního dotazníku pro další vylepšení navržených materiálů.

Dotazník zahrnoval čtyři otázky pro závěrečnou tvorbu statistiky přijetí vyzkoušené inovované úlohy a pro vyjádření názoru na princip zpracování prostřednictvím jednotkových testů a automatické kontroly v systému GitLab. Dotazník měl anonymní formu, student byl požádán o jeho vypracování po zpracování úlohy s tvorbou třídy zaměstnance dle návodu v prezentaci. První otázka zjišťovala míru složitosti zkušební úlohy, od jejího získání až po nahrávání do systému GitLab. Druhá otázka byla zaměřena na spokojenost s kontrolou úloh pomocí jednotkových testů. Třetí otázka umožnila vyjádření míry spokojenosti studenta s využitím systému GitLab. Ve čtvrté otázce se student mohl vyjádřit formou dodatečných připomínek, případně zde bylo možné uvádět chyby, které v inovované úloze (nebo návodu) objevil. U prvních tří otázek student vybíral z předem definovaných odpovědí, u poslední pak mohl využít textového pole – vyplnění bylo dobrovolné.

V rámci LMS Moodle bylo dále obsaženo společné fórum, viditelné všem studentům, kteří kurz „Programování v jazyce C/C++“ absolvovali. Zde mohl student vložit nové téma diskuze – především spojené s případnými komplikacemi při vypracování úlohy. Každý student měl také možnost reagovat na dříve vytvořená témata diskuze.

Po vytvoření podkladů pro zpracování úlohy a možnosti následného vyplnění dotazníku byla vybraná inovovaná úloha zadána studentům jako téma týdenního samostudia. Tato úloha nahradila první úkol programování v jazyce C++, který byl součástí probíhajícího kurzu.

8.2 Námět k úpravě přehlednosti testů a formátu dokumentace

Na základě zpětné vazby k inovovaným úlohám jiného kurzu programování, jmenovitě k úlohám se zaměřením na algoritmizaci a využití datových struktur v jazyce C, byly obecně jednotkové testy (kterých se využívá pro kontrolu dosažení požadované funkcionality úlohy) nedostatečně informativní a místy málo přehledné. Tyto připomínky pocházely převážně z řad studentů, kteří nové úlohy daného kurzu ve výuce již používali. Původně navržené zpracování jednotkových testů, využívající nástroje MinUnit, uživateli zobrazilo pouze symbol tečky (v případě úspěšného provedení) nebo chybovou hlášku v situaci neúspěšného vykonání testu. Řešením, které zajišťuje vyšší přehlednost, byla úprava souboru *minunit.h* tak,

že si student při vypracování úlohy určuje úroveň výpisu informací o prováděných jednotkových testech. Původní výpis je nyní u všech inovovaných úloh jazyka C++ rozšířen o volitelné uvedení názvu testovací sady, ve které se přiřazení provádí, na standardní výstup při spuštění testů. Rovněž je k dispozici nejvíce informativní varianta, která studentovi dále vypíše obsah každého prováděného přiřazení (tj. například která členská funkce vybrané třídy je porovnána s danou očekávanou hodnotou).

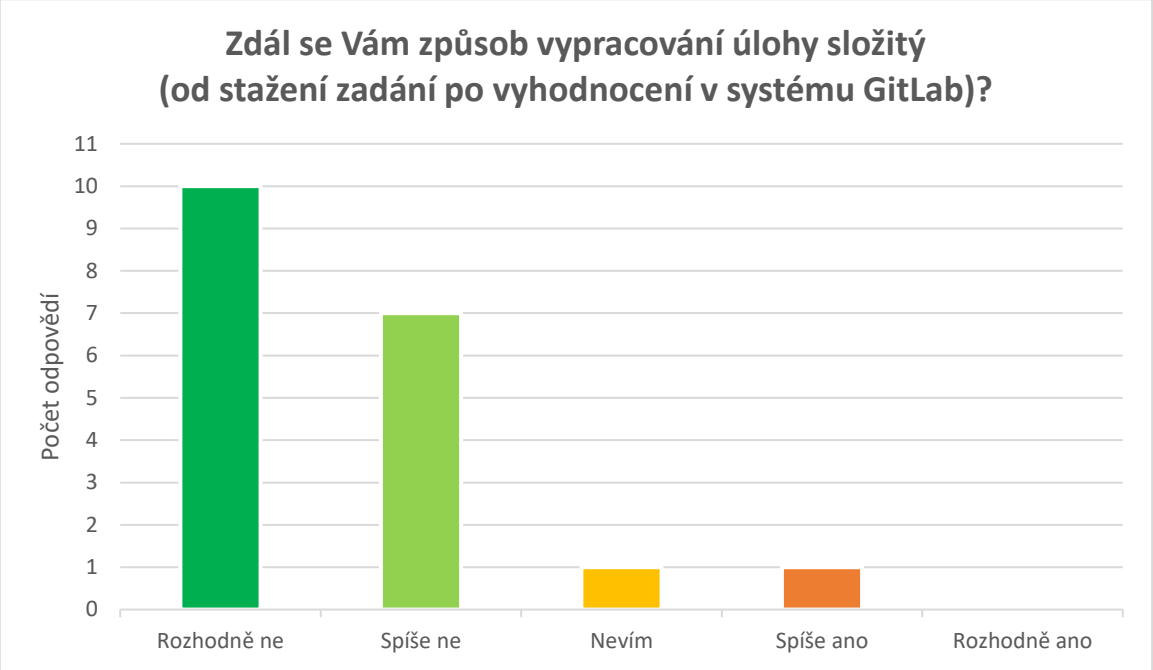
Dalším případným konfliktem v přímočarosti navržených zadání úloh je testování studentem zpracovaných součástí očekávanými hodnotami, které nejsou přímo specifikovány v návodu úlohy nebo v dokumentaci zdrojového kódu. U všech zpracovaných úloh jsem tedy zkontroloval a v určitých bodech doplnil dokumentaci ve formátu generátoru Doxygen, která je umístěna nad deklarací členských vlastností a funkcí využívaných tříd. U vybraných testovacích sad v souboru *tests.cpp* všech úloh jsou v současném provedení rovněž umístěny dodatečné bloky dokumentace, které jsou zaměřeny na shrnutí průběhu více komplexních testů (jaká součást se testuje a čím). V průběhu kontroly souvislosti dokumentace s navrženými testy jsem rovněž objevil dosud opomenuté gramatické chyby v anglicky psaných popisech, ty jsou nyní opraveny. Záhlaví každého souboru, kde jsou mimo jiné umístěny informace o jménu souboru a autorství, v původní verzi obsahovalo diakritiku, jež je nyní kvůli kódování souborů odstraněna.

8.3 Výsledná statistika přijetí inovované úlohy

Studenti obdrželi zadání úlohy k vypracování počátkem dubna s předpokládaným dokončením zpracování v průběhu semestru, který se uzavíral v polovině května. V době uzavření hodnocení samostatné práce studentů a přidělení bodů k udělení zápočtu byl anonymní dotazník vyplněn devatenácti studenty, z nichž jej někteří rozšířili také o připomínky k úpravám úlohy či náměty na další vylepšení. Společné fórum v systému LMS Moodle, pro vkládání případných dotazů spojených s řešením úlohy, nebylo žádným studentem využito. V průběhu řešení úlohy studenty jsem sledoval úpravy jednotlivých členů v systému GitLab a s dokončením sběru dat z dotazníku vypracoval statistiku přijetí úlohy.

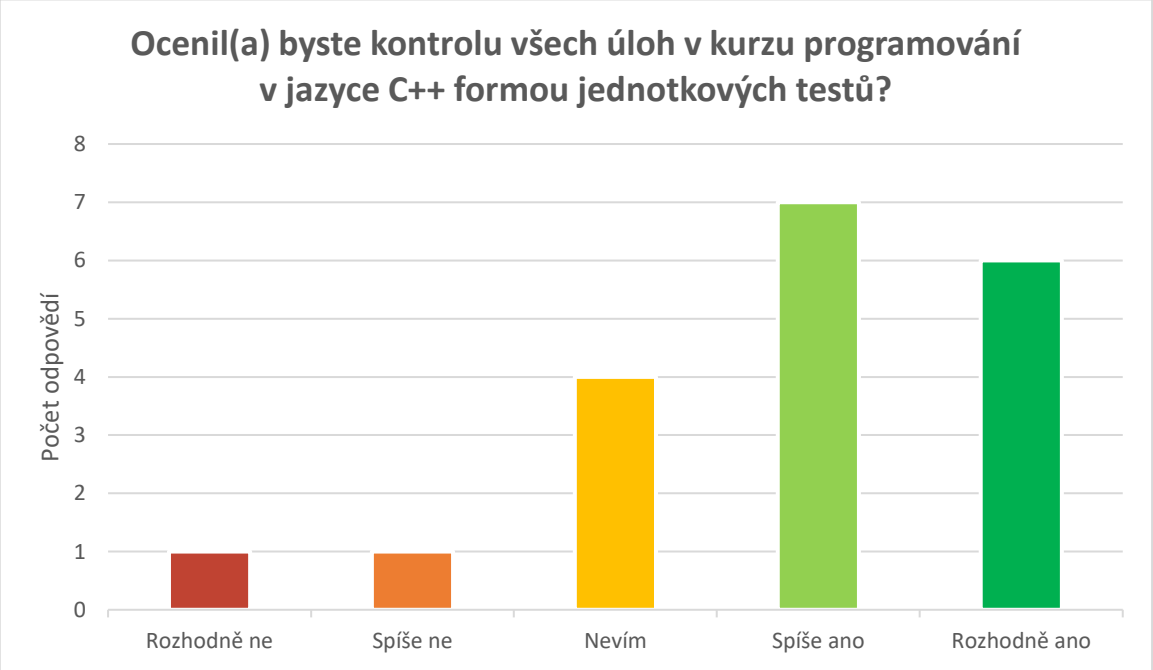
První dotaz byl cílen na zjištění míry složitosti při celkovém vypracování úlohy, které zahrnuje způsob získání zadání ze systému GitLab, následnou přípravu vývojového prostředí a

implementaci řešení dle návodu s opětovným nahráváním do zdrojového profilu úlohy studenta v systému GitLab, kde lze pozorovat výsledky kontrolní procedury.



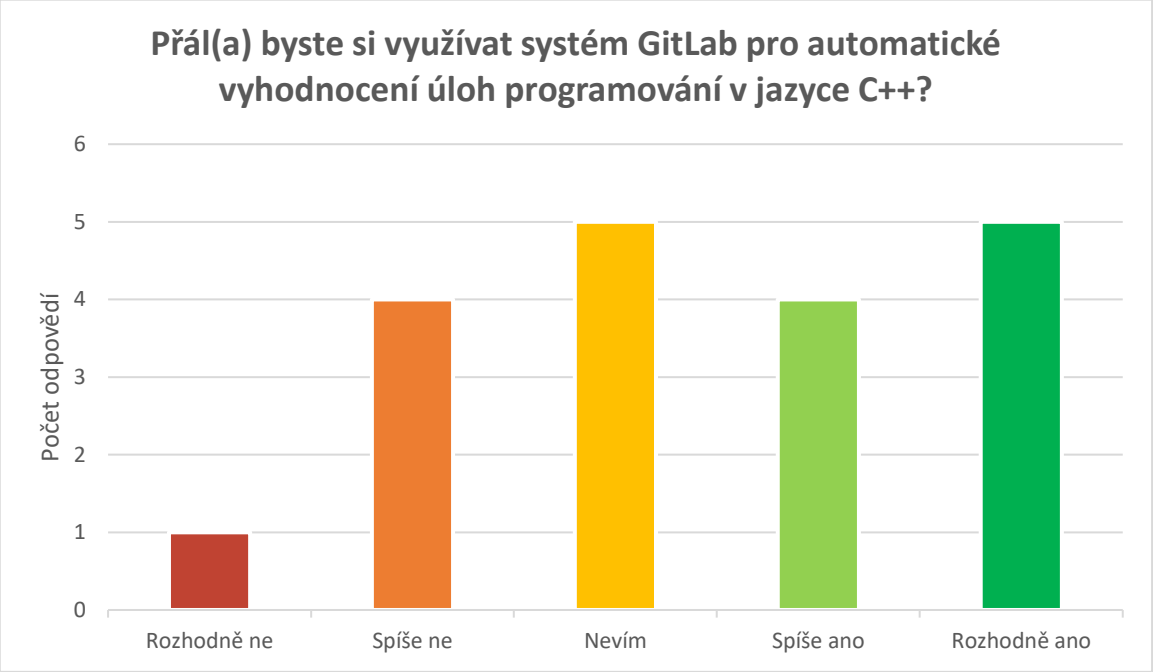
Graf 1: Statistika anonymního dotazníku – otázka první

Druhá otázka sledovala míru spokojenosti s využitím jednotkových testů pro ověření úspěšné implementace řešení dle zadání úlohy.



Graf 2: Statistika anonymního dotazníku – otázka druhá

Třetí otázka zjišťovala spokojenost studentů s využitím systému GitLab, odkud museli získávat šablonu se zadáním úlohy a kam také své řešení navraceli pro spuštění automatické kontroly.



Graf 3: Statistika anonymního dotazníku – otázka třetí

Navzdory omezenému počtu získaných odpovědí lze vypozorovat převažující názor, že zpracování poskytnuté úlohy (včetně využití jednotkových testů a systému GitLab) nebylo nadměrně složité, studenti spíše oceňují nový způsob kontroly implementace formou jednotkových testů a většinou, i když ze získaných dat nepříliš přesvědčivě, zastávají neutrální či spíše kladný názor na využití systému GitLab pro automatickou kontrolu zpracování úloh programovacího jazyka C++.

Při studentském zpracování úlohy prostřednictvím systému GitLab jsem rovněž využil sekce pro vkládání námětů ke zpracování, jež je označena *Issues*. Ve dvou případech použití jsem využil této sekce pro připojení komentáře za účelem poskytnutí nápovědy k řešení tak, aby

student dosáhl úspěšného splnění jednotkových testů a zjistil, ve které části zdrojového kódu byla obsažena chyba.[32]

Čtvrtou otázkou dotazníku, která byla dobrovolná, měl student možnost sdělit připomínky spojené s průběhem zpracování úlohy, případně zmínit nalezené chyby v zadání. Deset z devatenácti zúčastněných studentů připojilo k dotazníku i komentář. Zmíněné náměty k vylepšení obsahovaly poznámky k vytvořené prezentaci s návodem úlohy, ale také motivaci k odstranění nadbytečného řádku zdrojového kódu v souboru *employee.hpp*. Důkladná dokumentace ve formátu generátoru Doxygen se zdála u takto jednoduché úlohy jednomu z dotazovaných studentů příliš podrobná. Další připomínky zahrnovaly, mimo jiné, také nenalezení žádných chyb v úloze, ocenění způsobu zpracování úlohy nebo návodu k řešení (který studentům dříve nejasné ovládání systému GitLab nebo základ programování v C++ vysvětlil), také se objevily zmínky kladného přijetí jednotkových testů nebo systému GitLab obecně.

[32] Způsob využití GitLabu, konkrétně sekce *Issues*, tak odpovídá způsobu jeho použití některými zprostředkovateli MOOC v rámci distanční kontroly zdrojového kódu, viz 4.4.

ZÁVĚR

Při výběru obsahu inovovaných úloh jsem využil odborné literatury, odkud jsem problematiku zahrnutou v osmi vytvořených úlohách vyhledával dle míry složitosti takovým způsobem, aby se student pozvolným tempem seznámil se syntaxí a základními principy objektově orientovaného programování v jazyce C++. Počínaje první úlohou, kde se student setká s tvorbou třídy a instance objektu, v nově navržených úlohách dále poznává způsob přetěžování operátorů a skládání tříd, aplikuje pravidla dědičnosti pro vytváření odvozených tříd, také ví, jak lze využít výjimek a zjednodušit práci prostřednictvím součástí knihovny STL. Úlohy obsahují rovněž interpretaci řešení problematiky Jezdcovy procházky, využití binárních operací pro ukládání členské vlastnosti třídy, návrh členských funkcí třídy pro obsluhu jednoduchého databázového souboru knihovny SQLite a kupříkladu také tvorbu aplikace hry života dle pravidel, která jsou založena na principu buněčného automatu. Uplatněním navrženého přístupu k vypracování úloh poznává student také rozdíly programovacích jazyků C a C++.

Zhotoveny jsou dvě verze každé z úloh – celkové řešení (které slouží jako vzorové vypracování) a šablona se zadáním, jež student dle návodu rozšiřuje. Návod je specifický pro každou úlohu, je vypracován v českém a anglickém jazyce, u některých úloh je zahrnuta i ukázka formou úryvku zdrojového kódu. Inovované úlohy obsahují dokumentaci kódu, která se drží struktury generátoru Doxygen pro popis funkcionality součástí, společně s návodem tak informuje o očekávané podobě zpracování úkolu. Soubory obsahující zdrojový kód jsou na závěr formátovány nástrojem Clang-Format, vždy je tedy dodržena konzistentní podoba zápisu.

Pro vyhodnocení zpracované implementace jsou navrženy testovací scénáře a obsažené jednotkové testy, které porovnávají očekávané hodnoty se studentem navrženým řešením konkrétní úlohy (tj. s členskými vlastnostmi, funkcemi, případně hodnotami proměnných). Testy jsou vytvořeny pomocí nástroje MinUnit, jejich průběh si může student během zpracování zobrazit a své řešení dle aktuálních výsledků dále rozvíjet. Kromě platných hodnot jsou jednotkovými testy v určitých případech kontrolovány i ty neplatné, čímž se ověřuje ošetření výjimek a robustnost navržených součástí. O chybném provedení testu je student informován, zjišťuje, která součást jeho řešení nesplňuje požadavky.

Kontrolní proces studentem vypracovaného řešení úlohy je obstarán v systému GitLab, odkud student své zadání získává a poté i odevzdává k vyhodnocení. U všech osmi úloh, shodným způsobem u šablon vzorového vypracování i zadání úlohy, jsou navrženy konfigurační soubory pro sestavení projektů pomocí CMake a Qt Build Suite, je zohledněno vypracování pomocí operačního systému Windows i Linux. Konfiguračním souborem součásti GitLab CI/CD je naplánována automatická kontrola při pozměnění souborů projektu s úlohou, po odevzdání řešení je tak zkontrolována úspěšnost sestavení projektu, dále jsou vyhodnoceny výsledky jednotkových testů úkolu a pomocí nástroje Valgrind je ověřeno správné uvolnění paměti před ukončením programu. Tyto výsledky jsou studentovi dostupné k nahlédnutí, své řešení poté může bez omezeného počtu možných změn dále upravovat.

Každý student má v systému GitLab zhotoven vlastní soukromý profil s úlohou, takže nemůže nahlédnout do řešení svých kolegů ani je pozměnit. Průzkum spojený s možnostmi realizace kontroly v systému GitLab mi poskytl inspiraci k využití podstránky chybových zpráv (Issues), odkud může cvičící rozšířit automatickou kontrolu označením klíčové části zdrojového kódu, kde pozoruje chybu, tím tak poskytnout osobní zpětnou vazbu k řešení úlohy vybraného studenta přímo v systému GitLab.

Na základě zpětné vazby k úpravě výpisu informací o prováděných jednotkových testech je nyní nástroj MinUnit upraven tak, aby podporoval více informativních úrovní (ve smyslu četnosti výpisu textových řetězců na standardní výstup), mezi kterými si student může v průběhu zpracování vybírat. Pro snazší pochopení testovaných součástí úlohy jsou testovací soubory *tests.cpp* místy rozšířeny o dodatečnou dokumentaci. V případě méně přehledných testovacích sad je tak zdůrazněno použití konkrétních implementovaných funkcí a výběr očekávaných hodnot.

Jedna z inovovaných úloh programování v jazyce C++ byla poskytnuta studentům jakožto náplň samostatné práce, čímž si vyzkoušeli získávání úlohy, styl jejího zpracování a odeslání s kontrolou správnosti – své dojmy poté studenti vyjádřili v anonymním dotazníku. Průběh zpracování úlohy studenty jsem sledoval a v případě nejasností poskytl zpětnou vazbu. S ukončením zkoušky úlohy jsem vypracoval statistiku, ze které (s ohledem na nižší počet zúčastněných studentů) vyplývá, že většina dotazovaných považovala nabídnutou úlohu za jednoduchou, spíše oceňuje využití jednotkových testů pro kontrolu implementace úloh v jazyce C++ a raději využije systém GitLab pro automatickou kontrolu správnosti úloh.

SEZNAM POUŽITÉ LITERATURY

[1] STROUSTRUP, Bjarne. The C++ programming language. 4th ed. Upper Saddle River: Addison-Wesley, 2013. ISBN 978-0-321-56384-2.

[2] KENT, Jeffrey A. C++ bez předchozích znalostí: [průvodce pro samouky]. Brno: Computer Press, 2009. ISBN 978-80-251-2411-6.

[3] Identifiers. Cppreference.com [online]. San Francisco: Cubbi, 2019, 7 Nov 2019 [cit. 2020-03-26]. Dostupné z: https://en.cppreference.com/w/cpp/language/identifiers

[4] PECINOVSKÝ, Rudolf a Miroslav VIRIUS. Objektové programování 2: učebnice s příklady v Turbo Pascalu a Borland C++. Praha: Grada, 1996. ISBN 80-716-9436-3.

[5] GADDIS, Tony. Starting out with C++: From Control Structures through Objects. 8th edition. Boston: Pearson, c2015. ISBN 978-0-13-376939-5.

[6] SUTTER, Herb a Andrei ALEXANDRESCU. C++: 101 programovacích technik. Brno: Zoner Press, 2005. ISBN 80-86815-28-5.

[7] CHUA, Hock Chuan. Object-Oriented Programming (OOP) in C++: OOP Basics. In: Ntu.edu.sg [online]. Singapore: Nanyang Technological University, c2017, May, 2013 [cit. 2020-03-26]. Dostupné z: https://www3.ntu.edu.sg/home/ehchua/programming/cpp/cp3_OOP.html

[8] Polymorphism. In: Cplusplus.com [online]. cplusplus.com, c2000-2020 [cit. 2020-03-26]. Dostupné z: http://www.cplusplus.com/doc/tutorial/polymorphism/

[9] SCHWENK, Allen J. Which Rectangular Chessboards Have a Knight's Tour? Mathematics Magazine [online]. Kalamazoo (Michigan): Western Michigan University, 1991, s. 325-332 [cit. 2020-03-26]. DOI: 10.2307/2690649. ISBN 0025-570X. ISSN 0025570X. Dostupné z: http://www.jstor.org/stable/10.2307/2690649

[10] PARBERRY, Ian. An efficient algorithm for the Knight's tour problem. Discrete Applied Mathematics [online]. Denton (Texas): University of North Texas, 1997, 73(3), s. 251-260 [cit. 2020-03-26]. DOI: 10.1016/S0166-218X(96)00010-8. ISBN 0166-218X. ISSN 0166218X. Dostupné z: https://linkinghub.elsevier.com/retrieve/pii/S0166218X96000108

[11] Implementing Knight's Tour. MILLER, Brad a David RANUM. Problem Solving with Algorithms and Data Structures using Python [online]. Decorah, Iowa: Luther College, 2013, s. 1 [cit. 2020-03-26]. ISBN 1590282574. Dostupné z: https://runestone.academy/runestone/books/published/pythonds/Graphs/ImplementingKnightsTour.html

[12] HALDAR, Sibsankar. Inside SQLite [online]. Sebastopol (California): O'Reilly, c2007, s. 2-17 [cit. 2020-04-03]. ISBN 9780596550066. Dostupné z: https://books.google.cz/books?id=QoxUx8GOjKMC&printsec=frontcover&hl=cs&source=gbs_ge_summary_r&cad=0#v=onepage&q&f=false

[13] Cellular automaton. LifeWiki [online]. San Francisco: conwaylife.com, 2019, 25 August 2019 [cit. 2020-03-26]. Dostupné z: https://www.conwaylife.com/wiki/Cellular_automaton

[14] Conway's Game of Life. LifeWiki [online]. San Francisco: conwaylife.com, 2019, 30 December 2019 [cit. 2020-03-26]. Dostupné z: https://www.conwaylife.com/wiki/Conway%27s_Game_of_Life

[15] MYERS, Glenford J. The Art of Software Testing [online]. 2nd edition. Hoboken (New Jersey): Wiley, c2004 [cit. 2020-03-26]. ISBN 0-471-46912-2. Dostupné z: http://barbie.uta.edu/~mehra/Book1_The%20Art%20of%20Software%20Testing.pdf

[16] LANGR, Jeff. Modern C++ Programming with Test-Driven Development: Code Better, Sleep Better. Dallas (Texas): The Pragmatic Bookshelf, c2013. ISBN 978-1-937785-48-2.

[17] PASTOR, David Siñuela. MinUnit: Minimal unit testing framework for C. In: GitHub [online]. San Francisco: GitHub, c2020 [cit. 2020-03-26]. Dostupné z: https://github.com/siu/minunit

[18] Features. Doxygen [online]. Netherlands: Dimitri van Heesch, c1997-2018 [cit. 2020-03-26]. Dostupné z: http://www.doxygen.nl/manual/features.html

[19] Documenting the code. Doxygen [online]. Netherlands: Dimitri van Heesch, c1997-2018 [cit. 2020-03-26]. Dostupné z: http://www.doxygen.nl/manual/docblocks.html

[20] Special Commands. Doxygen [online]. Netherlands: Dimitri van Heesch, c1997-2018 [cit. 2020-03-26]. Dostupné z: http://www.doxygen.nl/manual/commands.html

[21] Overview: Core Clang Tools. In: Clang 11 documentation: Using Clang Tools [online]. The Clang Team, c2007-2020 [cit. 2020-03-26]. Dostupné z: https://clang.llvm.org/docs/ClangTools.html

[22] Clang-Format Style Options. Clang 11 documentation: Using Clang Tools [online]. The Clang Team, c2007-2020 [cit. 2020-03-26]. Dostupné z: https://clang.llvm.org/docs/ClangFormatStyleOptions.html

[23] Google C++ Style Guide. Google Style Guides: C++ Style Guide [online]. San Francisco: GitHub, c2020 [cit. 2020-03-26]. Dostupné z: https://google.github.io/styleguide/cppguide.html

[24] About CMake. In: CMake [online]. Clifton Park (New York): Kitware, c2000-2020 [cit. 2020-03-26]. Dostupné z: https://cmake.org/overview/

[25] CMake 2.8.0 Documentation. CMake [online]. Clifton Park (New York): Kitware, c2000-2020 [cit. 2020-03-26]. Dostupné z: https://cmake.org/cmake/help/v2.8.0/cmake.html

[26] Add_compile_options. CMake [online]. Clifton Park (New York): Kitware, c2000-2020 [cit. 2020-03-26]. Dostupné z: https://cmake.org/cmake/help/latest/command/add_compile_options.html

[27] Qbs Manual: Introduction. Qt [online]. Helsinki: The Qt Company, c2020 [cit. 2020-03-26]. Dostupné z: https://doc.qt.io/qbs/overview.html

[28] Qbs Manual: cpp. Qt [online]. Helsinki: The Qt Company, c2020 [cit. 2020-03-26]. Dostupné z: https://doc.qt.io/qbs/qml-qbsmodules-cpp.html

[29] About Valgrind. Valgrind [online]. Valgrind™ Developers, c2000-2019 [cit. 2020-03-26]. Dostupné z: https://valgrind.org/info/about.html

[30] Valgrind User Manual: Memcheck: a memory error detector. Valgrind [online]. Valgrind™ Developers, c2000-2019 [cit. 2020-03-26]. Dostupné z: https://valgrind.org/docs/manual/mc-manual.html

[31] GitLab Docs: User Docs. GitLab [online]. San Francisco: GitLab, [2020] [cit. 2020-03-26]. Dostupné z: https://docs.gitlab.com/ee/user/index.html

[32] GitLab Docs: GitLab CI/CD. GitLab [online]. San Francisco: GitLab, [2020] [cit. 2020-03-26]. Dostupné z: https://docs.gitlab.com/ee/ci/README.html

[33] GitLab Docs: GitLab CI/CD Pipeline Configuration Reference. GitLab [online]. San Francisco: GitLab, [2020] [cit. 2020-03-26]. Dostupné z: https://docs.gitlab.com/ee/ci/yaml/README.html

[34] SANCHEZ-GORDON, Sandra a Sergio LUJÁN-MORA. MOOCs gone wild. Proceedings of the 8th International Technology [online]. Valencia (Spain): INTED, 2014, s. 1449-1458 [cit. 2020-04-01]. ISBN 978-84-616-8412-0. Dostupné z: http://desarrolloweb.dlsi.ua.es/moocs/moocs-gone-wild

[35] SIMS, Zach. Codecademy Pro Welcomes Our 100,000th Learner. Updates [online]. New York: Codecademy News, c2020, 20 February 2020, 2020 [cit. 2020-04-01]. Dostupné z: https://news.codecademy.com/codecademy-pro-welcomes-our-100-000th-learner/

[36] Github Issues. Codecademy [online]. New York: Codecademy, c2020 [cit. 2020-04-01]. Dostupné z: https://www.codecademy.com/articles/github-issues-cheat-sheet

[37] Mastering Issues. GitHub Guides [online]. San Francisco: GitHub, c2020, April 7, 2014 [cit. 2020-04-01]. Dostupné z: https://guides.github.com/features/issues/

SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK

B3/S23	vzkříšení buňky mající 3 sousedy / přežití buňky při 2 nebo 3 sousedech
CI/CD	Continuous Integration/Continuous Delivery (Deployment)
FAI	Fakulta aplikované informatiky
HTML	Hypertext Markup Language
LLVM	low level virtual machine
LMS	Learning Management System
MOOC	massive open online course
OER	Open Educational Resources
OOP	objektově orientované programování
PDF	Portable Document Format
Qbs	Qt Build Suite
QML	Qt Modeling Language
SQL	Structured Query Language
STL	Standard Template Library
TDD	test-driven development
UTB	Univerzita Tomáše Bati
VMT	Virtual Method Table
VPN	virtual private network
YAML	YAML Ain’t Markup Language

SEZNAM GRAFŮ

SEZNAM ÚRYVKŮ ZDROJOVÉHO KÓDU

SEZNAM PŘÍLOH

P I Class_structure – návod.pdf

P II Inovace úloh jazyka C++.zip

PŘÍLOHA P I: CLASS_STRUCTURE – NÁVOD.PDF

Návod ke zpracování první inovované úlohy v archivačním formátu PDF/A, který byl poskytnut studentům v rámci distanční zkoušky úlohy v reálné výuce.

PŘÍLOHA P II: INOVACE ÚLOH JAZYKA C++.ZIP

Komprimovaná složka obsahující zpracované úlohy programovacího jazyka C++.