Inovace úloh v kurzu Programovací metody

Michal Horák

Bakalářská práce
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:	**Michal Horák**
Osobní číslo:	**A17121**
Studijní program:	**B3902 Inženýrská informatika**
Studijní obor:	**Softwarové inženýrství**
Forma studia:	**Prezenční**
Téma práce:	**Inovace úloh v kurzu Programovací metody**
Téma práce anglicky:	**The Innovation of Assignments in the Programming Methods Course**

Zásady pro vypracování

1. Prostudujte možnosti realizace automatické kontroly správnosti kódu v systému Gitlab.
2. Navrhněte sadu úkolů pro kurz Programovací metody, které budou zaměřeny na procvičení jazyka C, implementaci algoritmů pro práci s textem a implementaci vybraných dynamických datových struktur.
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.

Forma zpracování bakalářské práce: **Tištěná/elektronická**

Seznam doporučené literatury:

1. WRÓBLEWSKI, Piotr. Algoritmy. Brno: Computer Press, 2015. ISBN 9788025141267.
2. KERNIGHAN, Brian W. a Dennis M. RITCHIE. Programovací jazyk C. 2. vydání. Přeložil Zbyněk ŠÁVA. Brno: Computer Press, 2019. ISBN 9788025149652.
3. PROKOP, Jiří. Algoritmy v jazyku C a C++. 2., rozš. a aktualiz. vyd. Praha: Grada, 2012. Průvodce (Grada). ISBN 9788024739298.
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. WRÓBLEWSKI, Piotr. Algoritmy: datové struktury a programovací techniky. Brno: Computer Press, 2004. ISBN 8025103439.

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. v.r.
děkan

UNIVERZITA TOMÁŠE BATI VE ZLÍNĚ
Fakulta aplikované informatiky
L.S.

prof. Mgr. Roman Jašek, Ph.D. v.r.
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 diplomové/bakalářské práce bude uložen v příruční knihovně Fakulty aplikované informatiky Univerzity Tomáše Bati ve Zlíně a jeden výtisk bude uložen u vedoucího práce;
- byl/a jsem seznámen/a s tím, že na moji 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

Michal Horák, v. r.
podpis diplomanta

ABSTRAKT

Cílem této bakalářské práce je navrhnout a realizovat sadu úkolů pro kurz Programovací metody, u kterých bude prováděna automatická kontrola správnosti vypracovaných řešení. Práce se zabývá tvorbou a implementací úkolů, které seznámí studenty s programováním v jazyce C, s využitím verzovacího systému Git a také s jednotkovými testy pro testování správnosti vypracovaných řešení.

Teoretická část je zaměřena na obecné seznámení s programovacími jazyky, podrobněji pak s jazykem C, a na nástroje pro tvorbu úkolů, jejich odevzdání a následnou kontrolu. Také jsou zde popsány některé aspekty výuky programování. Praktická část popisuje strukturu a obsah vypracovaných úkolů, způsob testování správnosti vytvořených řešení studenty, výsledky testování úkolů v reálné výuce a návrhy na potencionální úpravy úkolů.

Klíčová slova: Jazyk C, Git, Jednotkové testování, výuka programování

ABSTRACT

Goal of this work is design and implement a set of assignments for “Programming methods” course, where automatic check of correctness with GitLab system will be used. This work deals with creation and implementation of assignments, introducing students to basics of programming in language C, using version-control system Git and its basic functions and using unit tests for testing correctness of solutions created by students.

Theoretical part is focused on a general introduction to programming, in more detail with C language, and on tools for creating assignments, their submission, inspection and grading. Some aspects of teaching computer programming are also described here. Practical part describes structure and content of created assignments, methods of testing the correctness of student solutions, results of testing the assignments in real lectures with students and suggestions for potential improvements of assignments.

Keywords: Language C, Git, Unit testing, programming education

Rád bych poděkoval vedoucímu práce, Ing. Tomáši Dulíkovi, Ph.D., za rady a pomoc při tvorbě této bakalářské práce. Také bych chtěl poděkovat Ing. Tomáš Juřenovi za základní koncepci struktury úkolů a rady při tvorbě.

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

OBSAH

ÚVOD

Výuka programování může být někdy náročná práce. Učitel musí studentům představit danou problematiku, vysvětlit ji tak, aby jí porozuměli, poskytovat podporu při řešení zadaného úkolu, a nakonec úkoly správně opravit. Pokud je studentů více, třeba i z více tříd nebo skupin, může vyučující strávit opravováním příkladů i několik hodin denně a nedokáže zajistit stoprocentní správnost opravy a následného ohodnocení.

Cílem této práce je inovovat úlohy pro kurz Programovací metody, tedy vypracovat šablony a řešení úloh na základě zadání, zavést tzv. jednotkové testování pro snadnější ověřování správnosti řešení a následně využít systém GitLab pro jednoduché předávání zadání studentům a pro následnou kontrolu odevzdaných řešení.

Pro výuku je využit programovací jazyk C, který sice nepatří mezi ty nejjednodušší na naučení, ale získané znalosti lze využít i v jiných programovacích jazycích. Jedná se o jazyk bez automatického uvolňování paměti, uživatel tedy musí uvolnit paměť po použití určitých datových typů a struktur, což někdy nemusí být pro studenty jednoduché.

Pro testování správnosti řešení se využívá několik typů kontrol. Jedním z nich je jednotkové testování, které využívá funkce, u kterých kontroluje, zda při určitém vstupu funkce vrátí předpokládaný výstup. Taktéž dochází ke kontrole úkolu jako celku, potažmo výstupu programu po jeho spuštění, u kterého zároveň dochází i ke kontrole práce s pamětí.

Pro odevzdání a závěrečnou kontrolu se využívá verzovací systém Git a webový repozitář GitLab. Na tomto repozitáři student získá zadání a odevzdává vypracované řešení, na kterém se provádí výše zmiňované testování.

I. TEORETICKÁ ČÁST

1 PROGRAMOVACÍ JAZYK

„Programovací jazyky jsou jazyky sloužící k tvorbě počítačových programů (programování). Programování je proces algoritmizace dané úlohy, tj. vytváření postupu, jenž vede k řešení dané úlohy.“ [1]

1.1 Typy a dělení

Způsobů, jak rozdělit programovací jazyky, je celá řada. Dají se dělit podle způsobu překladu, jejich funkcionality nebo například stáří. Také je lze dělit podle způsobu jejich využití nebo z jaké rodiny jazyků pocházejí, případně jejich určení.

1.1.1 Vyšší a nižší

Nižší programovací jazyky jsou ty, ve kterých se pro psaní kódu využívají téměř přesně příkazy, které odpovídají příkazům na procesoru. Nedochází proto k dodatečné změně kódu a vykonává se to, co uživatel napsal. Díky této skutečnosti jsou závislé na procesoru nebo na rodině procesorů a jejich instrukční sadě. Jelikož mají instrukční sady omezený počet příkazů, musí tvůrce kódu programovat i základní funkce, což vede k velmi složitému kódu. Na druhou stranu lze výsledný program velmi dobře optimalizovat a využívat funkce, ke kterým by vyšší programovací jazyk neměl přístup. Jako příklad lze uvést programovací jazyk Assembler. [1]

Vyšší programovací jazyky jsou srozumitelnější než nižší jazyky a nejsou závislé na procesoru a jeho příkazech. Samotný kód se ovšem musí převádět do strojového kódu, k čemuž slouží tzv. kompilátor, případně interpret.[1] Příkladem vyššího programovacího jazyka může být C#, Pascal nebo PHP.

1.1.2 Imperativní, logické a funkcionální

Imperativní, taktéž procedurální programovací jazyky jsou prakticky všechny jazyky, které je dnes běžně používají. Programátor vytváří posloupnost příkazů neboli algoritmus, který stanovuje postup řešení nějakého problému nebo úlohy. Tyto posloupnosti instrukcí se následně jedna po druhé provádí. [1]

Logické jazyky se od ostatních liší tím, že se neimplementuje samotný proces řešení daného úkolu, pouze výsledek popsaný pomocí logických výroků, ke kterému se má program dostat. Tyto jazyky jsou používány například v oborech pracujících s umělou inteligencí. [1]

Funkcionální programovací jazyky popisují vše pomocí funkcí, tedy program je pouze složitou soustavou funkcí. Tyto jazyky jsou blíže matematice než například imperativním jazykům. [1]

Logické a funkcionální programovací jazyky lze zařadit do jedné skupiny – deklarativní jazyky. Taktéž některé jazyky mají možnost využívat postupy z jiné skupiny, jako například imperativní Python, ve kterém lze využít i funkcionálního programování. [1, 2]

1.1.3 Interpretované a kompilované

Interpretované programovací jazyky jsou takové jazyky, u kterých dochází k překladu až v momentě, kdy dochází ke spuštění samotného programu. Ke spuštění je potřeba interpret, proto je většinou pomalejší, ovšem oproti kompilovaným jazykům lze kód přenášet mezi zařízeními, jelikož není vázán na danou instrukční sadu. [1, 2]

Kompilované jazyky jsou překládány k tomu, aby mohly být spuštěny. Oproti interpretovaným jsou rychlejší, ke spuštění není třeba interpret a také programátor neposkytuje přímo zdrojový kód, který by šel měnit nebo do něj jinak zasahovat. Také ale mají většinou vyšší nároky na formální správnost kódu [1, 2] a jeho implementace bývá většinou složitější.

1.1.4 Další možné dělení

- **Objektové a neobjektové –** Rozlišuje, zda daný programovací jazyk podporuje objektově orientované programování, tedy tvorbu různých částí kódu jako objekty a práci s nimi. Platí nepsané pravidlo, že cokoliv, co lze vytvořit objektově, lze vytvořit i bez objektů a naopak.
- **Moderní a zastaralé –** Toto dělení je subjektivní, a ne všichni s ním mohou souhlasit. Jako moderní lze považovat například ty, jenž jsou dnes hojně používány, jako je Java, C# nebo Python. Zastaralé zase mohou být třeba jazyky jako Pascal nebo Fortran, které dříve byly populární, ale dnes již existují lepší alternativy. Také jsou zde jazyky, které mohou být považovány za zastaralé, například C nebo Assembler, ovšem dodnes se používají a v některých případech nemají lepší alternativu.
- **Dělení podle účelu** – Programovací jazyky bývají většinou zaměřeny na nějakou oblast nebo platformu a používat ho jiným způsobem nemusí být ideální, někdy dokonce téměř nemožné. Například některé jazyky jsou vytvořeny a uzpůsobeny pro tvorbu webů a webových aplikací, proto pokusit se s nimi o programování embbeded systému nemusí být ideální, pravděpodobně i nemožné.

1.2 Příklady jazyků

Každý programovací jazyk má své specifické vlastnosti, které určují, pro jaké využití se nejlépe hodí. Některé jazyky jsou limitovány na specifické použití, pomocí jiných lze ale vytvářet software teoreticky pro cokoliv, i když to nemusí být vždy tak jednoduché, jako u jazyků určených pro tento daný typ problému. Zde jsou uvedeny některé z nejvíce používaných a známých jazyků.

- **Assembler –** Jazyk symbolických adres je nízkoúrovňový programovací jazyk, který pro psaní programů využívá přímo instrukce procesoru. Umožňuje tvorbu vysoce efektivního kódu, ovšem je velmi komplexní a implementaci vyžadují i ty nejjednodušší operace.
- **C++ –** Jedná se o původně rozšíření jazyka C o další funkcionalitu, jako je například objektové programování a funkce s tím spojené. Jazyk má širokou škálu využití, hlavně pro tvorbu aplikací, u kterých je potřeba co největší výkon a efektivita.
- **C#** – Jazyk C# je založený na jazycích C++ a Java, kterému je velmi podobný. Byl vytvořen pro tvorbu webových, ale i klasických aplikací. Je multiplatformní a také většinou využíván s frameworkem .NET, který obsahuje celou řadu dalších rozšiřujících funkcí.
- **Java –** Jeden z nejpoužívanějších programovacích jazyků na světě. Programy mohou být spuštěny na odlišných platformách, a to díky tomu, že jsou spouštěny pomocí JVM, který kód přizpůsobí danému zařízení. Využívaný je v systému Android, i když zde pomalu začíná ustupovat jazyku Kotlin.
- **PHP –** Skriptovací programovací jazyk zaměřený především na tvorbu webových stránek, potažmo jejich back-end části. Má proto velkou podporu různých nástrojů a funkcí, například pro využití databázových systémů.
- **Python –** Programovací jazyk zaměřený na jednoduchost, díky jeho syntaxi, a efektivitu programování, jelikož je stručný a také je odolný vůči chybám. Lze ho využít například na získávání dat z webů nebo třeba pro strojové učení.

1.3 Programovací jazyk C

Jazyk C je vyšším jazykem s prvky nižšího jazyka, někdy považován přímo za jazyk nízkoúrovňový. Jedná se také o procedurální jazyk, někdy nazývaný také imperativní nebo příkazový. Také je to jazyk kompilovaný, proto je nutný kompilátor pro přeložení kódu do

spustitelné aplikace. Oproti dalším jazykům z rodiny C není jazyk objektově orientovaný, což ovšem nemusí být zápor, a neobsahuje tzv. garbage collector, tedy algoritmus na automatické uvolňování již nepoužívaných prostředků z paměti. Uživatel proto musí spravovat paměť sám, čímž se na jednu stranu při správném použití zvýší efektivita kódu, na druhou ale zvýší náročnost vývoje kódu. [2,3]

1.3.1 **Historie**

Jazyk byl vyvinut Denisem Ritchiem, který spolu s Brianem Kerninghan v roce 1972 vytvořily první návrh jazyka pro programování v systému UNIX. Syntaxe byla postavena na jazyce B, z čehož vychází i název jazyka. Jedním z důvodů vytvoření jazyka C bylo přepsání zdrojového kódu systému UNIX z assembleru, ke kterému má jazyk C blízko, a také proto, že existující jazyk B na tento úkol nestačil. V pozdějších letech došlo k velkému rozšíření, díky implementaci do architektury IBM PC. Také začala vznikat standardizace jazyka organizací ANSI, jako jsou normy C89, C99 a nejnovější C11. [4]

1.3.2 **Charakteristika**

„Program v jazyce C se nezávisle na své velikosti skládá z funkcí a proměnných. Funkce obsahuje příkazy určující, jaké výpočetní operace se mají provést, a v proměnných se ukládají hodnoty používané během výpočtu.“ [5]

1.3.2.1 Klíčová slova

Jedná se o vyhrazené identifikátory, které slouží k identifikaci datových typů, příkazů, či jiných konstrukcí jazyka. [6]

auto	double	inline *	sizeof	volatile
break	else	int	static	while
case	enum	long	struct	_Bool *
char	extern	register	switch	_Complex *
const	float	restrict *	typedef	Imaginary *
continue	for	return	union	
default	goto	short	unsigned	
do	if	signed	void	

Obrázek 1 Klíčová slova jazyka C [6]

Pro ukládání hodnot, ze kterými program pracuje, slouží tzv. proměnné. Ty mají datový typ, který určuje, jakým způsobem jsou data v proměnné uložena, jméno, které umožnuje s touto proměnnou v programu nějak pracovat a samotnou hodnotu odpovídající datovému typu

proměnné. Mezi jednotlivými datovými typy lze pomocí funkcí přecházet a měnit jednu na druhou neboli je přetypovávat. Lze proto například z typu pro znak *char* získat číselnou hodnotu *int* a naopak, převádět z čísla typu *long* na typ *int* atd. Datové typy lze také „skládat" dohromady, kdy vytvoří datovou strukturu mající několik datových typů v jednom.

Příkazy určují činnost, která se má provést a jakým způsobem. Definují základní operace, jako jsou cykly nebo podmínky, případně operace, které základní operace narušují a mění jejich běžné chování. Příkazy lze skládat dohromady a tvořit bloky kódu, které vykonávají nějakou funkci. Tyto bloky mohou mít podobu funkcí, které lze následně volat s jiných částí kódu, aby provedly funkci v nich obsaženou a případně i nějakou hodnotu vrátily.

1.3.2.2 Struktura kódu

Program v jazyce C je tvořen posloupností deklarací, což znamená, že je složený s deklarací funkcí, typů a proměnných. Jedna z těchto funkcí musí být pojmenována jako *main()*. Jedná se o vstupní bod programu – program vždy začíná v této funkci, protože je volána systémem a ne uživatelem, a většinou v ní také končí. [6] Návratový typ funkce je integer, který slouží i indikaci, jakým způsobem program skončil. Pokud vrátí funkce například 0, program skončil korektním způsobem, pokud 1, program skončil s chybou. Jedná se však o konvenci, která nemá na běh programu vliv. Vstupní parametry funkce jsou nepovinné, nemusí být tedy žádné, nebo je lze využít pro vstupní parametry programu *int main(int argc, char *argv[])*, kde první parametr je číslo s počtem vstupních parametrů a druhý je dvourozměrné pole symbolů s jednotlivými vstupy. Program má vždy alespoň jeden vstupní parametr, kterým je cesta ke spustitelnému souboru programu.

1.3.2.3 Knihovny

Program je možné rozšířit o další funkce pomocí tzv. knihoven. Většinou se jedná o samostatné části kódu, které poskytují nějakou funkcionalitu, případně celou sadu funkcionalit. Knihovny mohou být vytvořené nějakým uživatelem nebo může být poskytována jako standartní knihovna.

Samotná knihovna se většinou skládá ze dvou částí. První je hlavičkový soubor (*.h*), který obsahuje deklarace funkcí, které lze následně v programu využít. Může být deklarována jako tzv. prototyp funkce, tedy pouze záhlaví funkce implementované v .c souboru, nebo může obsahovat i implementaci, většinou ovšem velmi krátkou a jednoduchou, jako například sečtení dvou čísel a vrácení výsledku. Druhou částí je zdrojový soubor (*.c*), který implementuje

funkcionalitu deklarovanou v hlavičkovém souboru, tzv. veřejné funkce. Může ovšem obsahovat i funkce, které v hlavičkovém souboru nejsou, tzv. privátní funkce nebo proměnné. Také lze pro všechno využít pouze zdrojový soubor, který bude zahrnovat i to, co by běžně bylo součástí hlavičkového souboru.

Pro využití v kódu se musí knihovna deklarovat. K tomu slouží příkaz *#include,* po kterém následuje název knihovny buď v uvozovkách ("knihovna.h") jako relativní cesta k dané knihovně vzhledem k adresáři s kódem, nebo ve znacích větší/menší (<knihovna.h>) jako relativní cesta ke standartnímu umístění. První způsob se používá většinou pro vlastní knihovny nebo pro knihovny třetích stran, kdežto druhý se používá na předdefinované knihovny, jako jsou knihovny STD s funkcemi například pro vstupy/výstupy nebo přidání dalších datových typů, jako například datový typ boolean (<*stdbool.h*>).

1.3.3 **Možnosti a využití**

Využívá se pro software, kde je potřeba co nejvyšší efektivita, jako ovladače, jádra operačních systémů nebo pro zařízení s limitovanými prostředky, tedy embbeded systémy nebo mikropočítače a mikrokontrolery. V současnosti je poměrně rozšířený, lze ho využít například pro programování mikrokontrolerů, jako je Arduino. Také je v něm napsána část operačního systému Windows.

1.3.4 **Nástroje pro sestavování a překlad projektu**

Každý program napsaný v jazyce C musí být přeložen, aby mohl být vytvořen spustitelný soubor. Existuje několik způsobů, jak proces překladu spustit, a to pomocí příkazového řádku, z IDE nebo pomocí například nástroje *make*. [6]

Při použití příkazové řádky se sestavuje příkaz, který začíná tím, jaký překladač použít. Každý překladač má následně k dispozici parametry, kterými je například možné specifikovat, v jakém jazyce je kód napsán, název výsledného spustitelného souboru, nebo všechny soubory, které jsou pro běh programu potřeba. [6] Pří větším počtu souborů se tento postup stává neefektivním.

Pokud využijeme vývojové prostředí IDE, je tento proces zjednodušený tím, že se o většinu potřebných úkonů stará vývojové prostředí samo, případně dokáže opravit některé konfigurační chyby. Samozřejmě uživatel může upravovat některé parametry a tím dále specifikovat požadovaný výsledek. Způsob, jakým překlad probíhá a co se má překládat a sestavit je většinou specifikováno v konfiguračním souboru spjatým se zvoleným překladačem.

Poslední možností je využití nástroje pro automatizaci překladu, například utility Make. Jedná se o pomocný program spuštěný v příkazovém řádku a pracujícím se soubory *make-file*, které obsahují instrukce pro vytvoření daného spustitelného souboru a které by byly normálně zadávány při každém překladu ručně. [6] Podobné utility mohou být také integrovány a spouštěny i v IDE.

1.3.4.1 CMake

CMake je open-source a cross-platformní nástroj sloužící k sestavování programů, převážně určený pro jazyky C a C++, i když jej lze využít i u jiných jazyků. Původně byl určen pro vytváření různých dialektů souboru Makefile, do kterého se zapisují instrukce pro program GNU Make. [7] Pro konfiguraci využívá textový soubor CMakeList, ve kterém se specifikuje minimální požadovaná verze CMake nástroje, název projektu, soubory projektu, se kterými má pracovat a které má ignorovat, nastavení kompilátoru, výstupní cesty a další nastavení. Také dokáže v jednom projektu pracovat s více spustitelnými programy, kdy každý má své vlastní parametry, nastavení a případně i zdrojové soubory.

Do prostředí Qt Creator, ve kterém byly vytvářeny a testovány jednotlivé úkoly tohoto kurzu, se musí tento nástroj dodatečně přidat a nakonfigurovat, ovšem postup je jednoduchý a dostupná Qt dokumentace[1] tento proces dostatečně popisuje. V případě projektů vytvořených v rámci této bakalářské práce je využit pro sestavení projektů během tvorby řešení studenty.

1.3.4.2 Qbs

Stejně jako předchozí CMake je nástroj Qbs určen pro sestavování softwarových projektů, a je dostupný na více platformách. Byl vyvíjen společností Qt, proto je tento nástroj implementován i v prostředí Qt Creator. Konfigurační nástroj využívá značkovací jazyk QML, který využívá pro svou funkcionalitu programovací jazyk JavaScript. [8] Samotná konfigurace se ukládá do souboru ve formátu .qbs, který určuje například název projektu nebo parametry kompilátoru. Stejně jako CMake dokáže rozdělit projekt na více aplikací, se kterými lze do jisté míry nezávisle pracovat.

[1] Dostupné na https://doc.qt.io/qtcreator/creator-project-cmake.html

Tímto nástrojem se v tomto projektu řeší sestavování na straně serveru pro automatickou závěrečnou kontrolu a ohodnocení, teoreticky ho lze využít i pro sestavování při tvorbě řešení.

V roce 2018 bylo oznámeno opuštění vývoje a nahrazení Cmake systémem. [9] Díky tomu, že se jedná o open source projekt, byl vývoj převzat komunitou, která nástroj stále aktivně vyvíjí. [10]

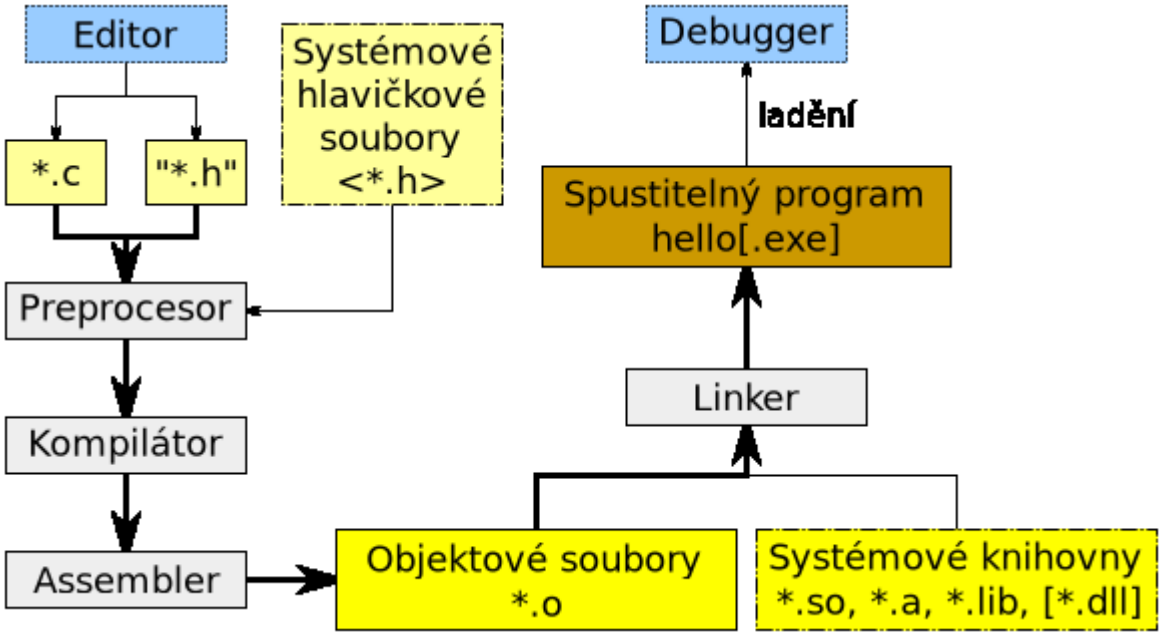
1.3.5 **Překladač, kompilátor**

Protože jazyk C je jazykem kompilovaným, musí se před spuštěním napsaný kód zkompilovat do binárního kódu, který následně bude dekódován a spuštěn na procesoru. Před samotnou kompilací je ovšem zapotřebí kód upravit do vhodného stavu. Kód proto prochází několika procesy, jejichž výsledkem je spustitelný soubor.

1.3.5.1 Proces překladu

Samotný proces je rozdělen na několik částí. První probíhá tzv. preproccessing, při kterém se nahrazují makra. Například obsah souborů nebo knihoven, které jsou přidány pomocí příkazu *#include,* se vloží do okolního kódu, čímž vznikne jeden velký soubor, obsahující veškerý kód. Dále probíhá překlad do objektových souborů (.o). Tyto soubory obsahují strojový kód, avšak adresy externích funkcí a proměnných jsou stále reprezentovány symboly. Proces se skládá ze dvou částí. Prvním je překlad výstupu preprocesoru do jazyka symbolických instrukcí, prováděný kompilátorem. Druhým je následný překlad těchto instrukcí do binárního kódu, k čemuž slouží program assembler, někdy zaměňován za stejnojmenný programovací jazyk. [11, 12]

Po vytvoření objektových souborů následuje tzv. linkování. Při vytváření objektových souborů kompilátor neví, zda existují volané funkce nebo kde se nachází. Úkolem linkeru je nahradit název funkce voláním adresy v paměti, kde se daná funkce nachází. Dochází ke spojení jednotlivých objektových souboru a knihoven do jednoho celku, ze kterého se následně vytvoří samotný spustitelný soubor, což je i finální krok překladu. Spustitelný soubor se liší podle platformy a někdy i jazyka, v případě OS Windows se jedná o soubor s koncovkou .exe. [12, 13]



Obrázek 2 Schéma překladu jazyka C [14]

1.3.5.2 GCC

GCC neboli GNU Compiler Collection je soubor překladačů určených pro řadu programovacích jazyků, především pro jazyk C a C++. Původně byl určen pro operační systém GNU, později byl portován i na další platformy, především UNIX systémy. [15]

Vývoj je součástí projektu GNU, která má za cíl vylepšit překladač pro GNU a jeho Linux varianty. Snaží se rozšířit GCC na další architektury a platformy za cílem poskytnout optimalizovaný překladač a přilákat další vývojáře, kteří zajistí další vývoj a testování dalších funkcí. [16]

Tato kolekce je využívána jako překladač při vyhodnocování řešení úkolů na straně serveru a lze ji využít i pro tvorbu samotných řešení v případě, že se student rozhodne pracovat na systému založeném na UNIX. Lze ovšem využít i alternativní překladače, jako je například Clang, jehož cílem je nabídnout náhradu za GCC.

1.3.5.3 MinGW

Kompilátor MinGW je minimalistické open source vývojové prostředí, primárně určené pro vytváření aplikací pro platformu Windows. Jedná se o port GCC, zahrnující kompilátory pro vývoj nejen v jazyce C, ale i v jazycích jako C++, ADA nebo Fortran. Kompilátor není závislý na knihovnách třetích stran, ale využívá knihovny, které jsou přímou součástí OS Windows. Oproti jiným portům GCC, jako je například Cygwin, nevyužívá pro fungování

emulaci POSIX standartu, která by mohla způsobit zhoršení výkonu aplikace, případně i nestabilitu. [17]

V kontextu této práce je určen pro překlad při tvorbě řešení zadaných úkolů na platformě Windows, ovšem opět lze využít i jiných alternativních kompilátorů.

2 PODPŮRNÉ PROGRAMY A NÁSTROJE

Pro vývoj software existují specializované nástroje, které celý proces ulehčují. Některé jsou zaměřeny na samotné psaní kódu, další pak na opravu případných chyb v něm nebo na jeho nahrávání a ukládání na webový server. I když k vývoji softwaru v podstatě žádné nástroje nejsou vyžadovány, ulehčují práci a také zlepšují kvalitu výsledného produktu.

Díky využití programovacího jazyka C pro tvorbu příkladů, který je poměrně rozšířený a zároveň se nejedná o exotický jazyk, je rozsah dostupných nástrojů pro podporu vývoje více než dostatečný vzhledem k řešené problematice.

2.1 Vývojové prostředí IDE

Vývojové prostředí neboli Integrated Development Environment je program, který je určený na vývoj software, k čemuž poskytuje nástroje ulehčující samotný proces vývoje. Vývojových prostředí je celá řada, většinou zaměřené na konkrétní programovací jazyk nebo rodinu jazyků, případně přímo platformu, pro kterou se program vyvíjí. Také jsou prostředí jednoduchá, poskytující pouze základní funkcionalitu, jako kompilátor, ale jsou i prostředí s velmi velkým výběrem dalších funkcí, jako je debugger, integrace s verzovacím systémem Git, tzv. našeptávač atd.

Pro účely tvorby tohoto kurzu bylo jako hlavní prostředí zvoleno IDE Qt Creator, ve kterém byly tvořeny úkoly, ovšem není požadováno a je možno využít i alternativy.

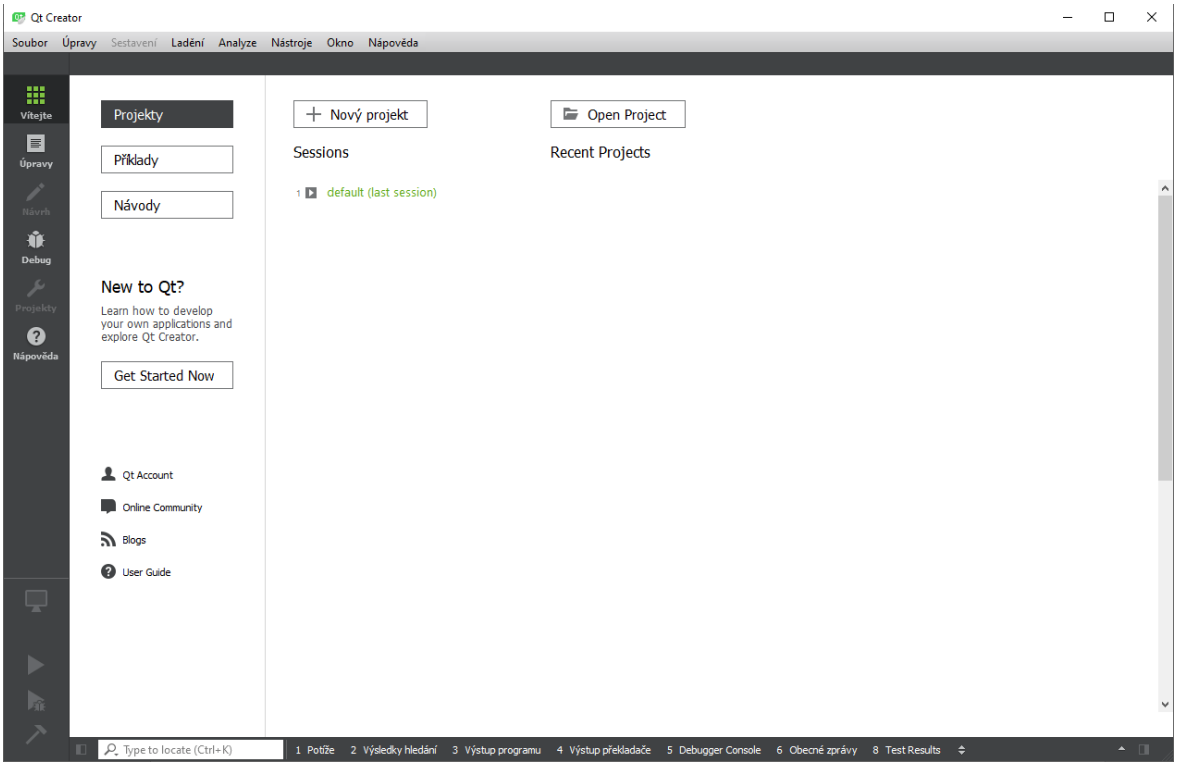
2.1.1 Qt Creator

Prostředí Qt Creator [18] je cross-platformní nástroj, který poskytuje nástroje pro tvorbu aplikací a softwaru pro klasické počítače, embedded zařízení a mobilní platformy. Prostředí je také dostupné pro platformy Windows, macOS a Linux. [19]

2.1.1.1 Uživatelské prostředí

Při prvním spuštění je prostředí otevřeno do *Welcome* módu, na kterém lze vytvářen nové projekty, otevírat již existující a možnost spuštění tutoriálů a projektů s příklady. Levý boční panel umožnuje výběr módu. Kromě možnosti *Welcome* je k dispozici mód *Edit*, který zobrazí hierarchii otevřeného projektu a obsah vybraného souboru, který lze upravovat. V tomto módu také probíhá samotné programování. Dalším módem je *Design*, který se používá k vytváření grafického rozhraní aplikace, pokud je samozřejmě potřeba. *Debug* je mód podobný módu Edit, ovšem navíc poskytuje nástroje pro testování, jako je analýza paměti nebo

profilování aplikace. Mód *Projects* složí pro nastavení sestavení aplikace při jejím kompilování a také pro nastavení vstupních parametrů a dalších možností při spuštění aplikace. Posledním módem je *Help*, který zobrazí dokumentaci prostředí, dostupnou také z webových stránek. Dalšími možnostmi na této boční liště je kit selector, který umožnuje výběr režimu, v jakém bude program spouštěn a také, jaká část projektu bude spouštěna. To lze využít například na spouštění jednotkových testů. Dále jsou zde možnosti pro spuštění vybrané sady, spuštění v režimu debuggování a tlačítko pro sestavení projektu. Výstupy těchto akcí lze zobrazit v panelech pro výstup ve spodní části okna. Nacházejí se zde výstupy kompilátoru, debugovacích nástrojů i programu samotného. [20]



Obrázek 3 Vývojové prostředí Qt Creator

2.1.1.2 Možnosti a využití

Jako ostatní vývojová prostředí slouží i Qt Creator ke tvorbě software, a to v různých jazycích a na různé platformy. Poskytuje k tomu sadu nástrojů, jako je integrace s verzovacím systémem, nástrojem pro tvorbu uživatelského rozhraní nebo emulátoru zařízení, pro které je software vyvíjen.

Protože obsahem tohoto kurzu je výuka programování, lze využít možnost prostředí spouštět více sad, nebo také částí programu, v jednom projektu, kdy jedna obsahuje samotný program a druhá knihovnu a program pro jednotkové testy.

2.1.2 Alternativní prostředí

I přesto, že pro tvorbu řešení i pro jejich vypracování bylo zvoleno prostředí Qt Creator, není nezbytné, aby bylo využíváno výhradně to. Existuje celá řada alternativních prostředí, které podporují stejné nástroje a funkcionalitu, které využíváme během tvorby a řešení úkolů. Jako příklad lze uvést prostředí Codelite [21] nebo Eclipse [22].

2.2 Verzovací systém

Jedná se o nástroj, který má za úkol uchovávat soubory a historii vyvíjeného projektu, umožnit distribuci jeho aktuální verze mezi vývojáře na něm pracujícím. Také může zajistit kontrolu případných editačních kolizí a jejich opravu.

V dnešní době existují dva druhy verzovacích systémů. Centralizované, které mají jedno centrální úložiště a využívají připojení typu klient-server. Mezi tyto systémy patří například SVN nebo CVS. Druhou variantou jsou systémy distribuované, kdy každý vývojář má vlastní verzi repozitáře, ze kterých se následně sestavuje výslednou verzi. Tyto systémy jsou většinou peer-to-peer. Jako příklad lze uvést Mercurial nebo Git, který se v této práci využívá. [23]

2.2.1 Git

Git [24] je distribuovaný verzovací systém dostupný jako open source a zdarma pro všechny uživatele. Je rychlý, díky provádění většiny operací lokálně, což poskytuje výhodu oproti centralizovaným systémům. Protože se jedná o distribuovaný verzovací systém, jsou k dispozici vícenásobné zálohy, kde má každý uživatel daného repozitáře lokálně uloženou svou kopii. Lze tak v případě znehodnocení souborů na serveru využít jakoukoliv z existujících záloh. [25]

2.2.1.1 Práce se systémem

Využívat systému Git lze několika způsoby. Lze využít příkazové řádky, do které zadáváme přímo příkazy, které následně systém vykoná a zobrazí výsledek. Dále lze také použít některé z grafických rozhraní, kde se samotnými příkazy téměř nepřijdeme do styku. Existují také vývojová prostředí, které mají systém integrovaný přímo.

2.2.1.2 Příkazy

Pro využití systému pomocí příkazové řádky se využívá sada příkazů, každá s určitou funkcionalitou. Jsou příkazy pro spuštění pracovní oblasti, tedy příkaz *clone*, který zkopíruje vybraný repozitář do lokální složky a příkaz *init*, jehož cílem je vytvořit nový repozitář nebo re inicializovat existující. Dále jsou příkazy pro práci se změnami, jako jsou *add* a *rm* pro přidání/odebrání souboru z indexu. Příkazy pro změny na repozitáři jsou například *branch*, pro zobrazení, vytvoření nebo smazání vývojových větví, *commit* pro zaznamenání změn na lokálním uložení a *merge*, který spojí dvě nebo více vývojových větví do jedné. Nakonec jsou tu příkazy pro nahrávání a stahování změn. Příkaz *fetch* slouží ke stažení objektů a referencí z jiného repozitáře, *pull* stáhne z repozitáře integruje s jiných repozitářem nebo lokální větví a *push* pošle změny na lokální větvi na hlavní repozitář.

2.2.1.3 Zabezpečení přenosu

Zabezpečení přenosu na úložiště lze dosáhnout několika způsoby. Pokud se jedná o veřejný repozitář, do kterého lze volně přispívat, zabezpečení většinou nepoužívá. Jestli je ovšem repozitář privátní, je potřeba zajistit, aby mohli přispívat pouze ti uživatelé, kteří na to mají právo, a to tak, že uživatel prokáže svou identitu tím, že se přihlásí do systému pomocí svého uživatelského jména a hesla. I když se jedná o celkem spolehlivou metodu ověřování, v kontextu této práce může být zdlouhavá.

Další možností je využití Single-Sign-On pomocí SSH protokolu. Ten zajistí, že se uživatel „přihlásí“ pouze jednou a následná komunikace se serverem z daného zařízení již probíhá bez dalšího přihlašování ze strany uživatele. Tohoto se dosáhne pomocí výměny klíčů, na základě kterých bude probíhat autentizace uživatele. Klíče se vytváření pomocí šifry RSA, která vytvoří dva klíče, privátní a veřejný, o stanovených parametrech. Veřejný klíč se potom poskytne serveru, se kterým chceme komunikovat a privátní se využije pro konfiguraci systému Git a služby SSH-agent. Při následném využívání systému Git a přístupům k repozitářům bude pro ověřování využíván protokol SSH. Pro navýšení zabezpečení lze využít tzv. passphrase, kdy pro využití protokolu SSH musí být zadáno uživatelské heslo. [26]

2.2.2 **Možnosti využití**

Verzovací systémy slouží hlavně pro vývoj a následnou údržbu velkých i malých projektů. Každý projekt má přiřazené vývojáře, kteří mají k projektu přístup. Když vývojář chce pracovat na nějaké funkcionalitě, stáhne nebo aktualizuje svou lokální verzi, provede v ní

změny a ty následně nahraje spolu s popisem změn zpět na repozitář. Ostatní vývojáři díky tomu pracují vždy s nejaktuálnější verzí daného projektu. Díky historii lze také případné změny zvrátit, pokud je to za potřebí.

Lze ho ale využít, jako v případě úkolů v této práci, i pro distribuci zadání úkolů pro programování a jejich následné nahrání pro kontrolu správnosti.

2.3 Webový repozitář

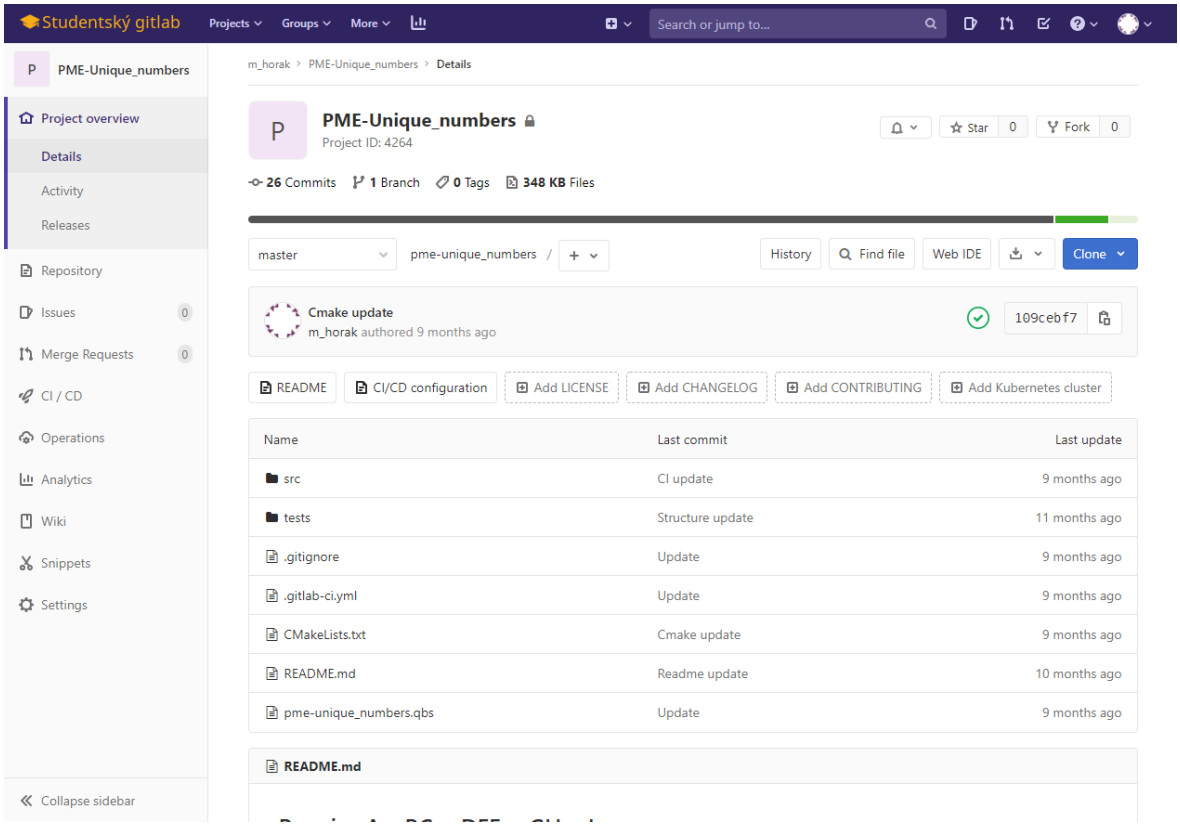
Možností ukládání souborů na internetu je celá řada. Může se jednat o obyčejné „hloupé" úložiště, kdy se soubory pouze přesunují, jako je cloud úložiště nebo FTP server. Existují ale i úložiště, které byly vytvořeny přímo pro účel vývoje a správy softwaru.

2.3.1 **GitLab**

GitLab je open-source software, tedy jádro programu je zdarma a open-source, ovšem je k dispozici i komerční verze, která je rozšířená o další nástroje a podporu. V jedné aplikaci poskytuje sadu nástrojů pro kompletní životní cyklus vývoje a nasazení produktu, tzv. DevOps. Oproti konkurenčním řešením je aplikace poskytována i pro použití na vlastních serverech, umožňující mít plnou kontrolu nad stavem a funkcemi aplikace bez nutnosti platit za navyšování velikosti úložiště nebo dodatečné funkce. [27]

2.3.1.1 Webové rozhraní

Jedná se o serverovou aplikaci, proto se přístup k ní provádí hlavně skrze webový prohlížeč. Repozitář podporuje více projektů naráz, které lze dále dělit do skupin a složek. Každý projekt má vlastní webovou stránku, na které má oprávněný uživatel přístup k samotným souborům daného projektu, které lze prohlížet a měnit, jeho nastavení a k nástrojům určeným pro testování a nasazení. Dále zde lze zobrazit historie změn v projektu, jako jsou změny provedené pomocí systému Git, také vývojové větvě projektu, případně jeho tagy a velikost. Taktéž jsou zde k dispozic nástroje pro analytiku, pro nahlašování problémů nebo pro ukládání částí kódu a textu, které lze dále sdílet mezi uživateli. Každý projekt má taktéž možnost, která umožní buďto stáhnout celý projekt komprimovaný v požadovaném formátu, získat adresu projektu pro využití verzovacího systému Git nebo stáhnout přímo vybraný soubor. Pokud se v projektu nachází soubor s názvem README.md, který většinou obsahuje popis projektu, zobrazí se tento soubor na úvodní stránce.

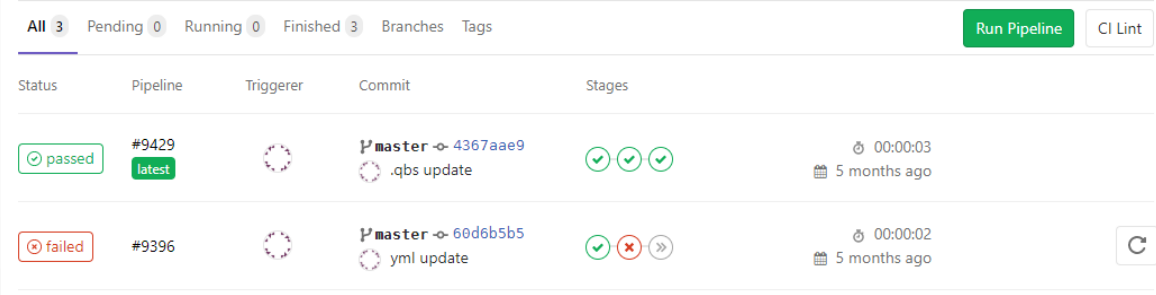


Obrázek 4 GitLab – Projekt Unique_numbers

2.3.1.2 Nástroj CI/CD

GitLab obsahuje celou řadu nástrojů pro podporu vývoje, jako nástroje pro analytiku projektů, dokumentaci pomocí wiki stránek nebo nástroje pro vývoj software pomocí „průběžné metodiky“ CI/CD.

Tento nástroj, značící název *Continuous integration* a *Continuous delivery/deployment*, je nástroj, který umožnuje v průběhu vývoje testovat vytvořené řešení a také ho připravit, nebo přímo uvést do ostrého provozu. CI je založena na automatickém spouštění skriptů, tedy na minimálním zásahu ze strany člověka. Zahrnuje průběžné sestavování a testování při každé nové verzi kódu, čímž potlačuje šanci na udělání chyby, která by mohly být přenesena do dalších verzí. Tento proces může probíhat několikrát za den, i při minimálních změnách v kódu. Zkratka CD může znamenat dvě věci. Buďto se jedná o *Continuous delivery*, kdy po provedení CI se aplikace připraví na její vypuštění do ostrého provozu, ale samotné vypuštění musí být provedeno manuálně. V případě *Continuous deployment* se vypuštění provádí také automaticky, tedy bez zásahu člověka. [28]



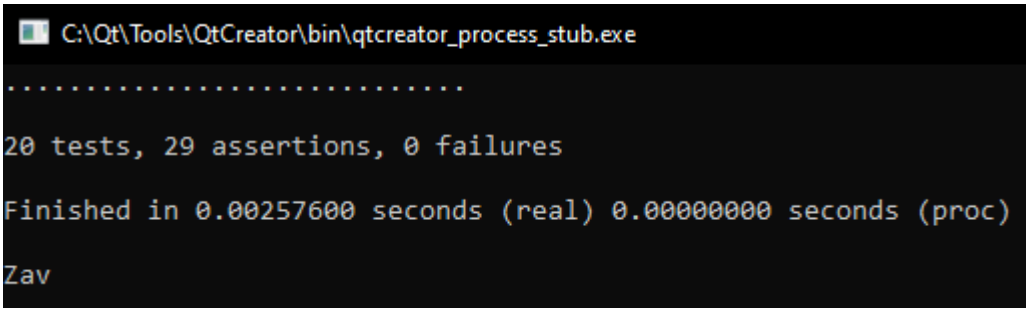
Obrázek 5 Pipelines nástroje CI/CD

2.4 Jednotkové testování

Tento způsob testování, v originále nazývaný Unit testing, slouží při vývoji software pro průběžné zkoušení částí kódu, které má za cíl ověřit, že přidávání dalších funkcionalit nebo refaktorizace kódu nezměnila způsob fungování dané funkce a že pro daný vstup funkce vrátí stále správný výstup. Doba, kdy se testy vytváření, závisí na zvolené metodice vývoje, například při vývoji řízeném testy TDD se nejdříve vytváří testy a až posléze se tvoří samotný kód.

2.4.1 Knihovna MinUnit

MinUnit je minimalistická knihovna pro jednotkové testování [29], dostupná pro jazyky C a C++. Poskytuje způsob pro vytváření a konfigurování testů, jejich rozdělování do testovacích sad („test suite“) a poskytnutí souhrnu ohledně množství spuštěných testů, jejich úspěšnosti a celkové době běhu.



Obrázek 6 MinUnit – příklad bezchybného výstupu z úkolu

Knihovna umožnuje testovat několika způsoby a ke každému poskytuje odpovídající funkci. *mu_check(condition)* provádí kontrolu zadané podmínky. Pokud je podmínka splněna, je test splněn, pokud ne, vrací se podmínka jako chybová hláška. Dále tu jsou testy *mu_assert(condition, message)*, která navíc v případě nesplnění podmínky vrátí v parametru zadanou zprávu a funkce *mu_assert_int_eq(expected, result)*, která spolu s variantou pro datový typ

double kontrolují, zda hodnota nějaké proměnné odpovídá očekáváné hodnotě zadané jako parametr.

2.5 Nástroj Valgrind

Jelikož jazyk C nemá garbage collector, který by se staral o uvolňování nepoužívané paměti, musí vývojář přiřazené prostředky kontrolovat a spravovat tak, aby nedocházelo ke zbytečnému vytěžování systémových prostředků. Pro debugování a ověřování paměti lze využít programu Valgrind, který je dostupný pouze pro operační systémy založené na UNIXu, tedy Linux nebo macOS. Pro použití na operačním systému Windows se musí pro lokální testování využít buď alternativa, nebo s posledními verzemi Windows 10 využít vestavěného subsystému WSL, který poskytuje plnohodnotné linuxové jádro umožňující běh tohoto softwaru.

Pro analýzu paměti slouží nástroj Memcheck, který při průběhu našeho programu kontroluje přidělenou paměť a po skončení testovaného programu vypíše informace o tom, jak se s pamětí zacházelo, potažmo jestli byla správně alokována a jestli byla všechna správně dealokována. Ke zjištěným únikům dodá i funkce, které s touto částí paměti pracovali. Poskytuje taktéž shrnutí, obsahující počet volání jednotlivých funkcí na správu paměti a celkový počet použité paměti. Jedním z volitelných nastavení je také hloubka analýzy paměti a množství informací, které budou uživateli zobrazeny. [30]

Součástí programu jsou i další nástroje, jako je Cachegrid pro profilování cache pamětí procesoru, Helgrind pro analýzu vláken a vícevláknových programů nebo Massif pro měření množství využité paměti typu halda. [30]

3 VYUČOVÁNÍ PROGRAMOVÁNÍ

Přestože je tato práce zaměřena na tvorbu příkladů pro programovací jazyk C a zadání úkolů bylo využíváno již v minulých letech, neznamená to, že se jedná o jediný možný způsob výuky programování.

Existuje celá řada způsobů, jak programování vyučovat. Ty jsou založené na určitých aspektech, jako je například zvolený programovací jazyk nebo způsob vyučování.

3.1 Volba jazyka

Existuje mnoho možností, v jakém programovacím jazyce se začít učit programovat. Jednou z možností je takový jazyk, který má jednoduchou syntaxi a jehož překlad umožňuje dobré zotavení ze syntaktických chyb. Jako příklad lze uvést programovací jazyk Pascal, který byl původně určen pro výuku strukturovaného programování. Má jednoduchou syntaxi a není case-sensitive, tedy nerozlišuje velká a malá písmena. V dnešní době již ovšem není v praxi příliš používán, proto lze pro výuku v tomto jazyce argumentovat tím, že sice lze zde předvést základy programování, ale z dlouhodobého hlediska a v případě, že se student rozhodne programovat jako jeho způsob obživy, bude se muset naučit i jiný, více používaný a v praxi využitelný programovací jazyk.

Další možností je využít rozšířenějšího programovacího jazyku, který je dnes hojně využíván. Toto má ovšem také své nevýhody. Jazyky, jako je například C# nebo Java, mají již pokročilejší strukturu, která může pro začátečníky a výuku základů být příliš složitá a nemusí být přímo jasné, co daný kód dělá.

3.2 Způsoby výuky

Způsobů, jak se učit programovat je celá řada. Volba závisí na samotném jedinci, který se chce učit programovat. Samozřejmě ve výsledku bude muset student projít praxí, kde se naučí pokročilejší techniky a styly programování využívané v praxi.

3.2.1 **Samoučení**

Asi nejběžnějším způsobem učení programování je výuka pomocí různých návodů a tutoriálů na internetu, kdy je člověk seznámen se základy a syntaxí zvoleného jazyka pomocí příkladů, na kterých autor předvádí nějakou funkcionalitu. Také se lze v případě, že dané problematice nerozumím nebo jsem ji nepochopil, zeptat buď přímo autora nebo lidí, kteří

se také učí pomocí daného materiálu, případně těch, kteří tuto problematiku ovládají a jsou ochotni pomoci ostatním.

3.2.2 Výuka ve škole

Dalším způsobem je výuka programování ve škole. Oproti samostatné výuce zde student nemá možnost volby, protože daný studijní program je po zapsání studenta pevně dán a náplň předmětů a programovací jazyky jsou většinou již vybrány. Toto ovšem může být také výhoda, protože student se setká s řadou různých programovacích jazyků, se kterými se pracuje rozdílně a jsou určené na řešení jiných problémů.

Také je zde oproti samoučení výhoda v tom, že je k dispozici vyučující, který dané problematice rozumí a orientuje se v ní. Proto může poskytnout studentovi rady v případě nějakého problému. Pro radu také může jít k jiným studentům, kteří mohou být na jiné znalostní úrovni, případně již se s problémem potýkali.

3.3 Zaměření úkolů

V úkolech vytvářených v rámci této bakalářské práce jsou úkoly zaměřeny buď na nějakou funkcionalitu daného programovacího jazyka, v našem případě jazyka C, nebo na obecnou problematiku programování, která je sice závislá na strukturách jazyka, ale princip lze aplikovat i na jiné jazyky.

Při vytváření úkolů pro výuku programování by tvůrce měl začínat s vysvětlením základních funkcí daného jazyka, dále předvést práci s proměnnými, funkcemi a případně i soubory. Po objasnění základů může přejít na pokročilejší funkce jazyka, v případě jazyka C například tvorba složitějších struktur a funkcí, práce s ukazateli nebo správa přiřazené paměti. Obsah úkolů se bude samozřejmě lišit podle toho, pro koho jsou určeny a pro jaký jazyk jsou tvořeny. Nemá například smysl učit objektově orientované programování, pokud samotný jazyk tento typ programování nepodporuje.

II. PRAKTICKÁ ČÁST

4 TVORBA ZADÁNÍ A VZOROVÝCH ŘEŠENÍ

Cílem této práce je inovovat úkoly v kurzu Programovací metody. Ty byly dříve zadány formou slovního popisu, podle kterého měl student vypracovat řešení, a to odevzdat na kontrolu do systému Moodle[2], což je systém v našem případně určený pro poskytování informací a úkolů k různým předmětům. Inovace úloh spočívá v tom, že všechny zadání budou převedeny do předloh, do kterých student doplňuje požadované funkce. Zadání spolu se šablonou získá na repozitáři GitLab, na který také bude odevzdávat vypracované řešení, které bude automaticky zkontrolováno.

Některé úkoly se prolínají i s předmětem ADS, proto byla vytvořena vždy jedna verze, která je sdílená mezi těmito kurzy. Tyto úkoly byly ovšem vytvářeny a testovány společně, i když mají uvedeného pouze jednoho autora.

4.1 Struktura úkolů

Úkoly se skládají z několika souborů, kdy každý poskytuje určitou funkcionalitu, jako konfigurace spuštění úkolů na serveru, konfigurace testů nebo nastavení kompilátoru. Projekt také zahrnuje samotné zdrojové soubory a případně i testovací vstupy.

Všechny soubory jsou psány v angličtině, jednak pro vyhnutí se speciálním českým znakům, obsahujícím háčky a čárky, které nemusí fungovat v některém z použitých nástrojů, a hlavně pro možnost výuky předmětu pro zahraniční studenty, kteří český, případně slovenský, jazyk neovládání.

4.1.1 Zdrojové soubory

Všechny soubory týkající se kódu, tedy hlavičkové a zdrojové soubory, mají stejnou strukturu. Obsahují záhlaví, implementaci nebo prototypy funkcí, případně mají přidané knihovny s dalšími funkcemi. Záhlaví a komentáře funkcí jsou napsané ve formátu podporovaném nástrojem pro tvorbu dokumentací Doxygen. Hlavička obsahuje název souboru, autora, datum vytvoření, stručný popis obsahu a funkce souboru a informace o autorských právech. Pokud se jedná o hlavičkový soubor, deklarace funkcí jsou komentovány taktéž ve formátu Doxygen a obsahují základní informace o jednotlivých funkcích.

[2] Dostupné na: https://moodle.org/

Všechny soubory jsou taktéž upraveny pomocí nástroje ClangFormat, který upravuje styl kódu, například maximální počet znaků na řádek nebo pozici složených závorek. Pro tuto úpravu se v tomto případě využívá Google C++ Style Guide[3], který v IDE Qt Creator existuje jako předdefinovaný. Studenti při tvorbě řešení tento styl dodržovat nemusí.

4.1.1.1 Soubor main.c

Každý nově vytvořený program napsaný v jazyce C začíná funkcí *main.* Ta se v našem případě nachází v souboru *main.c.* Většinou obsahuje funkce pro interakci s uživatelem, jako je výběrové menu, které následně volá funkce, které student musí implementovat. Akce prováděné ve funkci *main* jsou popsány, takže student může zjistit, co daný kód provádí, a tím si může v některých případech ulehčit tvorbu zadaných funkcí. Jediným souborem, který se přidává pomocí funkce *#include,* je hlavičkový soubor obsahující deklarace funkcí, které má student za úkol řešit. Soubor *main.c* je vytvořen tak, aby do něj student nemusel nijak zasahovat, což by mohlo vést k neúspěšnému vyhodnocení testů nebo přímo k nefunkčnosti řešení.

4.1.1.2 Hlavičkový soubor

V tomto souboru jsou obsaženy dopředné deklarace funkcí, které má student za úkol vytvořit. Komentáře funkcí pak obsahují popis funkce, její funkcionality a vstupní parametry, u kterých uvádí název, popisek a zda se jedná o proměnnou vstupní nebo výstupní. Popis také uvádí návratovou hodnotu, pokud nějakou má, a co se má v jakém případě vracet za hodnotu. Soubor také obsahuje odkazy na všechny ostatní knihovny a funkce, které jsou potřeba pro běh a tvorbu programu. Knihovny jsou přidávány tímto způsobem, aby nedocházelo k jejich zbytečnému přidávání na více místech, protože tento hlavičkový soubor je využíván i v ostatních zdrojových souborech.

4.1.1.3 Soubor s funkcemi

Funkce, které má student implementovat, se nachází právě v tomto souboru. Je zde odkaz na hlavičkový soubor s deklaracemi funkcí a samotné „kostry“ funkcí. V zadání jsou funkce implementovány tak, aby při spuštění testů v nedokončeném stavu nedocházelo

[3] Dostupné na: https://google.github.io/styleguide/cppguide.html

k nesprávnému ukončení testů, kdy jakékoliv další testy nebudou provedeny. Také obsahuje funkce, které jsou již implementované. Většinou se jedná o funkce pro uvolňování paměti, které mohou studentům způsobit nemalé problémy a nejsou cílem daného úkolu. Rozdělení funkcí a deklarací do separátních souborů je z toho důvodu, aby mohly být provedeny jednotkové testy.

4.1.1.4 Další knihovny

V pokročilejších úkolech jsou přítomny další podpůrné knihovny, které mají studentům usnadnit práci na samotných úkolech. První knihovnou je *ioutils.* Ta obsahuje funkce pro čtení znaků, textů nebo číselných hodnot ze standartního vstupu, což je většinou příkazová řádka nebo pro potřeby testování soubor s testovacím vstupem. Druhou knihovnou je *mymalloc*, která přidání funkce na zobrazování hodnot při alokovaní a uvolňování paměti, čímž zjednoduší hledání nesprávného přiřazování nebo uvolňování paměti. Oproti základním příkazům tyto hodnoty zobrazují, kolik paměti bylo právě alokováno, kolik je alokováno celkem a kolik se uvolňuje. Využití nachází tako knihovna u složitějších úkolů s většími datovými strukturami, jako je například vektor nebo binární strom.

4.1.1.5 Testovací zdrojové soubory

Pro testování správnosti implementace funkcí pomocí jednotkových testů slouží knihovna *minunit*. Ta lze být implementována i do samotného kódu, ovšem pro potřeby tohoto kurzu byl pro ni vytvořen zvláštní soubor, který provádí pouze ono testování. Protože se jedná o další soubor, který obsahuje funkci *main,* musí být tento soubor oddělen od hlavního *main.c,* aby nedocházelo ke kolizi a chybě překladače. Z tohoto důvodu jsou také funkce odděleny ve svém vlastním zdrojovém a hlavičkovém souboru. Funkce *main* v tomto testovacím zdrojovém souboru zahrnuje funkci na volání testovací suity, která následně volá funkce na testování funkci, a výpis zprávy o stavu dokončení jednotlivých testů, tedy úspěšnost, rychlost a počet testů.

Jednotlivé testy testují různé možné výstupy a vstupy, kde kontrolují, zda například pro vstup NULL funkce vrátí správnou návratovou hodnotu, nebo jestli funkce není v nekonečné smyčce. Testování tímto způsobem se provádí následně i u finální kontroly, takže student má přehled o tom, co se testuje a jakým způsobem.

4.1.2 Soubory se vstupy a výstupy

Některé úkoly mají pro běh nebo i testování vytvořené soubory se vzorovými vstupy. Může se jednat o čísla pro výběr nějaké možnosti z nabídky v konzoli, nebo přímo hodnoty určené pro nějakou funkci. Proto ve složce s projektem jsou přidány i tyto soubory, které mají studentovi předvést vzorový vstup, který by měla jejich implementace zvládat.

Kromě souborů se vstupy jsou také někdy přítomny soubory se vzorovými výstupy, které byly vytvořené ze vzorových řešení a ukazují, jak by měl výstup pro daný vstup vypadat. Toto porovnávání se provádí i při odevzdání úkolu jako další kontrola kromě jednotkových testů a analýzy paměti. Výsledkem je soubor *.diff*, který ukazuje rozdíly mezi předlohou výstupu a výstupem ze studentova řešení.

4.1.3 Nastavení pro sestavení

Pro sestavení projektů, obzvlášť v našem případě, je třeba programu pro sestavení specifikovat, s jakými soubory má pracovat a jakým způsobem. Každý náš projekt má dva soubory s tímto nastavením a každý pracuje s jiným softwarem, jiným způsobem a pro jiný účel. Při vytváření řešení studentem je používáno řešení pomocí CMake a jeho konfiguračního souboru CMakeList.txt. Při kontrole na serveru se využívá druhý systém, Qbs, který je složitější na konfiguraci, ale umožnuje nastavovat více parametrů. V případě nutnosti však student může student využít jakékoliv z daných možností pro vytváření sestavení, je však doporučeno pracovat pomocí řešení CMake.

Obsah obou souborů je z hlediska funkčnosti téměř totožný. Nastavují se možnosti pro kompilátor, jako je standart programovacího jazyka nebo jak se má prostředí chovat k varováním. Také specifikuje soubory pro daný spustitelný soubor, v našem případě pro samotný program a pro jednotkové testy. Zde také dochází k oddělení souborů obsahující funkci *main,* která může způsobit chybu překladu. Přidávají se také cesty, do kterých bude projekt sestavován a také speciální parametr pro případy, že projekt bude spouštěn na operačním systému vycházejícím s UNIXu.

4.1.4 Konfigurační soubory pro Git a GitLab

V neposlední řadě obsahuje projekt soubory pro nastavení verzovacího systému Git a pro spouštění a kontrolu řešení úloh na repozitáři GitLab. Systém Git si do svých inicializovaných repozitářů přidává vlastní složku se soubory a konfigurací, obsahující například jméno repozitáře a přítomné vývojové větvě, seznam existujících, změněných nebo smazaných

souborů projektu a další nezbytné komponenty. Samozřejmě tento soubor existuje jen pokud student využil pro stažení šablony systému Git.

Soubor *.gitignore*, který je zahrnut v zadání, obsahuje seznam souborů, které jsou při nahrávání změn pomocí sytému Git ignorovány a ponechány beze změny. Většinou se jedná o soubory se sestavením a uživatelskými nastaveními, které nejsou pro testování potřeba a jsou proto zbytečné.

Pro GitLab je zde soubor *.gitlab-ci.yml*, jehož úkolem je konfigurace pro testování a ověřování úkolu. Nastavují se zde parametry pro jednotlivé stupně celého procesu testování, jako je sestavení, jednotkové testování a testování porovnávání výstupů. Je zde zahrnuto i nastavení místa, kam se má projekt sestavit, taktéž kam se mají ukládat výstupy z testování, nebo i cesty k testovacím souborům, které jsou uloženy separátně, jakožto ochrana před možnou úpravou.

4.2 Obsah úkolů

Každý z úkolů je zaměřen na představení a procvičení jiné funkce jazyka C nebo způsobu programování využitelného v programování jako celku. Pro řešení zadaných úkolů se předpokládá, že student již někdy programoval, buďto přímo v jazyce C nebo jiném programovacím jazyce vycházejícím z jazyka C, neboť samotná koncepce úkolů a jejich implementace spoléhá na alespoň základní porozumění dané problematice a znalosti základních funkcionalit a možností jazyka C.

4.2.1 **Math_library**

První v řadě úkolů je zadání na vypracování matematických funkcí, které pracují s polem prvků, v tomto případě čísel, anebo vytváření sekvencí čísel. Cílem je procházet náhodně vygenerované pole v případě prvních funkcí a vytvářet čísla se zadaným vstupním parametrem, načež výsledky budou vráceny pomocí funkce *return*.

- **Functions_Minimax** – Vrací minimum a maximum z dodaných prvků. Jedná se funkci, která využívá funkcí pro hledání minima a maxima.
- **Functions_GetMin** – Funkce, která z poskytnutého pole prvků hledá ten prvek, který má nejmenší hodnotu. Využívá k tomu cyklus, u kterého porovnává prvky z pole vždy s nejmenším momentálně nalezeným, což je na začátku první prvek z poskytnutého pole.

- **Functions_GetMax** – Využívá stejného principu jako funkce pro hledání minima, pouze hledá hodnotu, která je v poli největší.
- **Functions_Factorial** – Hledání faktoriálu čísla zadaného jako vstupní parametr. Faktoriál čísla *n* je součin všech čísel, které následují po daném číslu četně, dokud se nedosáhne čísla 1. Výsledek následně vrací pomocí návratové hodnoty funkce.
- **Functions_Fibonacci** – Vrací číslo ve Fibbonacciho posloupnosti, tedy sekvenci, kdy číslo je rovno součtu předcházejících dvou čísel, na zadané pozici.

4.2.2 Equation

Úkol Equation je celý zaměřen na vytvoření algoritmu, který hledá taková unikátní čísla, aby platila rovnice *A*BC=DEF=GH*I,* kdy každé z písmen musí odpovídat právě jedinečnému číslu od 0 do 9. Počet, kolikrát má číslo povoleno se v rovnici objevit je nastavitelný, proto student musí brát v potaz i tuto proměnnou. Základem je pole o délce deset, kdy každý element odpovídá číslu od 0 do 9. V případě, že číslo bude použito v rovnici, inkrementuje se hodnota na dané pozici v tomto poli o jedna.

- **Functions_IsUnique** – Kontrola, zda číslo, které bude dosazeno do rovnice, je unikátní neboli jestli odpovídá zadanému rozsahu. Ve vstupních parametrech je číslo, u kterého testujeme jednotlivé číslice, zda již nejsou v rovnici obsaženy, pole s výskytem jednotlivých číslic a počet povolených výskytů. Vrací hodnotu *true* nebo *false* podle toho, zda je číslo jedinečné nebo ne.
- **Functions_FindEquation** – Funkce pro samotný algoritmus hledání rovnice. Jako vstupní parametry používá počet povolených výskytů a pole se strukturou *Equation*, skládající se z parametrů pro jednotlivé části rovnice a sloužící jako návratová hodnota, do které se ukládají rovnice potom, co byla, případně byly, nalezeny. Doporučeným postupem je postupně dosazovat čísla na jednotlivé pozice, dokud nedojdou platné možnosti. V tom případě se vrátíme na předchozí pozici a dosadíme jiné číslo. Tento proces opakujeme, dokud nenajdeme všechny platné rovnice. Samotná funkce vrací počet nalezených rovnic.

4.2.3 Histogram

Cílem úkolu je vytvořit histogram z daného souboru, tedy počet výskytů znaků abecedy a-z v načteném souboru. Program prochází celý textový soubor a zapisuje nalezený znak na odpovídající pozici v poli. Student má za úkol doplnit dvě funkce.

Úkol také obsahuje již hotovou funkci, která slouží na uvolnění dvourozměrného pole. Této funkce využívá pouze funkce *main*, která je již taktéž hotová, a tak se student nemusí touto funkcí vůbec zaobírat. Může ovšem posloužit jako nápověda pro budoucí úkoly.

- **Functions_LoadFile** – Funkce pro načtení souboru a následné uložení jeho obsahu do dvourozměrného pole znaků, které si lze představit jako pole obsahující jednotlivé věty nebo části textu o délce 256 znaků. Taktéž vrací hodnotu na základě toho, zda se povedlo otevřít soubor na dodané adrese.
- **Functions_Histogram** – Prochází zadaná pole a na základě ASCII hodnot znaků je ukládá do pole o velikosti 25, kdy pozice značí daný znak a hodnota na této pozici počet výskytů. Student také musí převést velká písmena na malá a ignorovat jakékoliv jiné znaky, jako mezery nebo speciální znaky.

4.2.4 CSV_file

Tento úkol je zaměřený na práci se soubory typu CSV, tedy tabulkový formát, kdy jednotlivé hodnoty jsou odděleny čárkami. Student pracuje s hodnotami získanými ze souboru, které ukládá do struktury a dále je zpracovává pro hledání určité hodnoty.

- **Person_LoadCSV** – Provádí načtení souboru a hodnoty v něm uložené převádí do dodaného pole struktur *tPerson,* která obsahuje hodnoty k uložení informací o fiktivní osobě, jako váha, výška a věk.
- **Person_GetAverageData** – Získaná data o osobách se v této funkci zprůměrují a hodnoty vrátí pomocí vstupních parametrů sloužících jako výstup. Průměrování se provádí sečtením všech hodnot z daného parametru, jako je například věk, a následné vydělení jejich počtem.
- **Person_Extremes** – Slouží k nalezení nejvyššího a nejnižšího věku mezi dodanými daty a následnému vrácení pozice neboli index oné osoby v rámci pole s ostatními osobami.

4.2.5 FSM_comment_remover

Jak lze z názvu vyčíst, tento úkol je zaměřen na odstraňování komentářů ze zdrojového kódu pomocí konečného stavového automatu. Konečný automat je založen na principu konečného množství stavů, mezi kterými lze přecházet na základě vytvořených podmínek, čímž dochází ke zpracování nějakých dat nebo informací a vytvoření odpovídajícího výstupu k danému

vstupu. Odstraňovač poznámek využívá tohoto automatu a prochází vstupní soubor obsahující kód v jazyce C nebo jiném programovacím jazyce, který má totožný styl komentování.

- **Functions_LoadStream** – Funkce pro nastavení a načítání vstupního a výstupního souboru nebo, pokud není vstupní a výstupní soubor zadán, nastaví vstup a výstup jako standartní vstup, potažmo výstup, tedy konzoli.
- **Functions_RemoveNotes** – Prochází jednotlivé řádky kódu získané ze souboru nebo ze standartního vstupu a na základě vytvořených pravidel pro jednotlivé komentáře, jako je jednořádkový a víceřádkový komentář nebo komentář ve stylu Doxygen, a ty vynechává z výstupního souboru vytvářeného během čtení a přepisu vstupního souboru. Pro správnou implementaci odstraňování komentářů je potřeba také brát v potaz jiné znaky, které mohou z domnělého komentáře udělat pouhý text nebo proměnnou, například v případě, že je komentář v uvozovkách jako hodnota proměnné.

4.2.6 **Floating_point**

V tomto úkolu je cílem vytvořit algoritmus na kontrolu zápisu čísel s tzv. plovoucí řádkovou čárkou. Syntaxe je stejná jako v jazyce C, tzn. poměrné volná, proto se například před tečkou nemusí nacházet číslo, pokud by dané číslo byla nula. Ovšem za tečkou, jako za znakem E, číslo být musí vždy, v případě znaku E může být před číslem znaménko.

- **Functions_CheckSyntax** – Obsahuje samotný algoritmus, který vyhodnocuje vstupní výraz a vrací, jestli je výraz platný nebo ne. Student může využít i konečného automatu pro tvorbu řešení, nejedná se ovšem o jedinou povolenou možnost, jak tento příklad řešit.

4.2.7 **Crossword_generator**

Generátor křížovek je úkol, ve kterém má student implementovat funkce potřebné pro vytvoření a vykreslení doplňovačky, která čerpá slova z dodaného souboru a na základě tzv. seedu, tedy inicializační hodnoty pro generátor náhodných čísel, pro kterou budou vygenerovány vždy stejná „náhodná“ čísla, budou vybrána vhodná slova a složena do hotové doplňovačky.

- **Wordlist_LoadWordlist** – Slouží k otevření vstupního souboru se slovy pro doplňovačku.

- **Wordlist_GetCharPosition** – Funkce pro nalezení pozice znaku v rámci vybraného slova, čímž se u kontrolovaného slova zjistí, zda slovo obsahuje písmeno, které koresponduje s klíčovým slovem a na jaké pozici se nachází.
- **Wordlist_GenerateKeyword** – Implementace vybrání klíčového slova, tedy slova sloužícího jako tajenka nebo také slova použitého k vybrání slov, které budou tvořit zbytek doplňovačky neboli horizontální slova k vertikálnímu klíčovému slovu.
- **Wordlist_ShowBoard** – Funkce na samotné vytvoření a zobrazení doplňovačky. Vstupem je pole pro vybraná slova, maximální hodnotu a vybrané klíčové slovo. Doporučeným postupem pro tuto funkci je vybrání tolika slov s takovými písmeny, aby odpovídaly tajence, a následně daná slova vypisovat do konzole s odsazením jednotlivých slov tak, aby po vypsání všech slov vzniklo vertikálně klíčové slovo, navíc ohraničené svislým lomítkem (znak '|').

4.2.8 **Phonebook**

Úkol Phonebook je zaměřen na načítání záznamů z .CSV souboru a jejich následnému seřazení podle zvoleného parametru. Každý záznam se skládá z jedinečného identifikátoru ID, jména a příjmení osoby, fyzické adresy a telefonního čísla. Úkolem studenta je vytvořit funkci pro načtení potřebných údajů ze souboru a přidat volání funkce *qsort*, která pomocí poskytnuté funkce pro porovnávání a nastaveného údaje, s jakou hodnotou má pracovat, porovná jednotlivé prvky a vrátí seřazené vstupní pole, jehož obsah je vypsán do konzole.

- **Person_LoadCSV** – Slouží pro načtení dat ze souboru a jejich rozřazení do správných proměnných v struktuře a jejich přidání do pole těchto struktur. Do struktury se data přidávají pouze v případě, že všechny údaje k dané osobě byly úspěšně načteny. Funkce vrací počet správně načtených osob, pokud se nepovedlo načíst soubor nebo výstupní pole, vrátí hodnotu -1.

4.2.9 **Vector**

Tento úkol je jedním z pokročilejších, u kterého je studentům představena práce se strukturou *Vector_t*. Zde je již potřeba se aktivně starat o přiřazenou paměť a udržování integrity paměti. Cílem úkolu je implementovat strukturu Vector, tedy dynamické pole, do kterého se přidávají prvky a na základě jejich počtu se automaticky alokuje další paměť. Funkce jsou zaměřeny na práci se strukturou, jako je kopírování obsahu mezi strukturami, hledání prvku na pozici, odstraňování a přidávání prvků nebo inicializace a mazání samotné struktury.

- **Vector_Create** – Vytvoření struktury *Vector*, které je přiřazena počáteční velikost pomocí vstupního parametru a alokační krok, tedy po jak velkých částech se bude přidávat dodatečná paměť při přidání prvku, na který již není přiřazená paměť dostatečně velká.
- **Vector_Copy** – Funkce na kopírování, která vytváří novou strukturu pomocí dat, které si zjistí z dodané originální, jako je alokační krok, originální velikost a počet volných míst pro hodnoty.
- **Vector_Clear** – Vyprázdnění vektoru vyplněním nul do proměnných s velikostí a volnými místy a nastavení hodnoty NULL pro proměnnou s položkami.
- **Vector_Length** – Vrací počet prvků ve struktuře, tedy počet zaplněných pozic z celkově přiřazené paměti, ne počet volných míst nebo velikost samotnou.
- **Vector_At** – Vrátí hodnotu vektoru na specifikované pozici v prvcích struktury. Vrací hodnotu *true*, pokud byl prvek na pozici nalezen, a také samotnou hodnotu pomocí vstupní hodnoty.
- **Vector_Remove** – Odstraní prvek na vybrané pozici a prvky, které následují onen mazaný prvek, posune tak, aby nezbyla žádná mezera. Provede se a vrátí hodnotu *true* v případě, že prvek byl nalezen, a hodnotu *false*, pokud nalezen nebyl a nic smazáno nebylo.
- **Vector_Append** – Přidává hodnotu získanou ze vstupního parametru na konec struktury, tedy za poslední přítomný prvek.
- **Vector_Contains** – Funkce sloužící k zjištění, zda je hodnota dodaná ve vstupním parametru přítomna ve vektoru. Pokud byla hodnota nalezena, vrací *boolean* hodnotu *true*, pokud ne, vrací hodnotu *false*.
- **Vector_IndexOf** – Hledá a vrací pozici prvku, který obsahuje hledanou hodnotu. Taktéž má parametr, který určuje pozici, od které se má začít hledat. Pokud je prvek nalezen, vrací se jeho pozice.
- **Vector_Fill** – Zaplní část vektoru od do pomocí stanovených vstupních parametrů hodnotou taktéž zadanou vstupní hodnotou. Vektor je přepsán od startovní pozice do konečné pozice. Pokud je konečná pozice větší než velikost samotného vektoru, přepisují se hodnoty pouze po poslední prvek. Jestli je startovní pozice za konečnou pozicí nebo koncem vektoru, začíná se od prvního prvku.
- **Vector_Destroy** – Kompletně vyčistí veškerý obsah struktury Vector a vrátí alokovanou paměť. Daný ukazatel na strukturu nastaví na NULL.

4.2.10 List

Cílem tohoto úkolu je implementovat strukturu List neboli uspořádanou kolekci datových položek, a funkce k ní přidružené. Hlavní strukturou je *List_t*, která obsahuje ukazatel na první položku a aktivní položku ze seznamu obsahujícího data. Tyto položky jsou taktéž struktury, obsahující samotná data a ukazatel na další položku v řadě. Úkolem je vytvořit implementaci základních operací s položkami a daty.

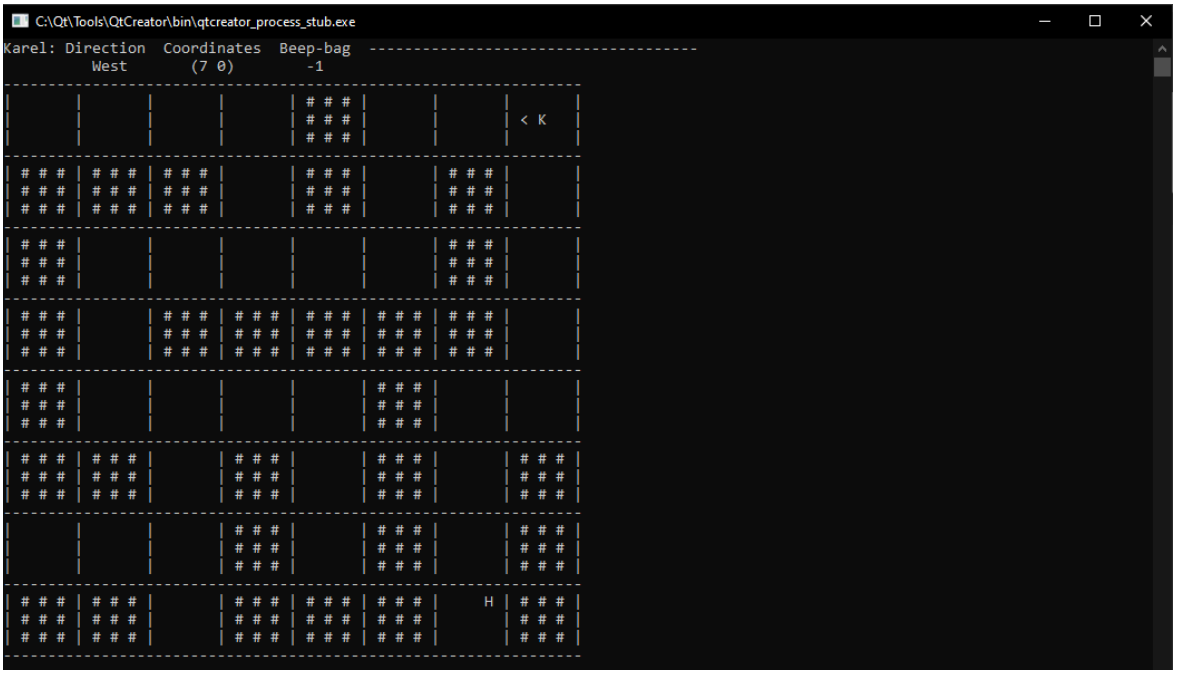
- **List_Init –** Inicializace hlavní struktury, tedy nastavení první a aktivní položky na hodnotu NULL. Vstupním parametrem je struktura, z kterou se bude pracovat.
- **List_Insert_First –** Vytvoří strukturu s daty získanými ze vstupního parametru, alokuje mu paměť a nastaví ho jako první prvek dodané hlavní struktury.
- **List_Copy_First –** Vrátí data do vstupního parametru ze struktury, na kterou ukazuje ukazatel značící, že se jedná o první položku v seznamu. Protože návratový typ je *boolean*, vrátí *true*, pokud čtení bylo úspěšné, *false* pokud nikoliv.
- **List_Delete_First –** Smaže první prvek. Pokud byl tento prvek zároveň aktivním prvkem, aktivní prvek bude nastaven na NULL.
- **List_Post_Delete –** Cílem funkce je smazat prvek, který následuje prvek aktivní, tedy položku, na který ukazuje ukazatel *next* v aktivní položce. Pokud není žádná položka nastavena jako aktivní, nic se neprovádí.
- **List_Post_Insert –** Vloží novou položku za aktivní prvek. Aktivnímu prvku se musí nastavit nový prvek jako další element v pořadí a novému prvku se nastaví prvek, který byl dříve jako další u aktivního, jako další. Když není aktivní prvek nastaven, neděje se nic.
- **List_Copy –** Vrací data z aktivního prvku, pokud má aktivní prvek přiřazenou hodnotu.
- **List_Actualize –** Aktualizuje data aktivního prvku, tedy nastaví proměnnou s daty na novou hodnotu. Funkce pracuje za předpokladu, že aktivní prvek má hodnotu.
- **List_Succ –** Posunuje aktivní prvek na další element v pořadí, tedy na ten, který má současný aktivní prvek nastaven jako další, za předpokladu, že existuje aktivní prvek.
- **List_Is_Active –** Vrací hodnotu *true*, pokud má aktivní prvek přiřazenou položku.

4.2.11 **Robot_Karel**

Tento úkol vychází ze programovacího jazyka vytvořeného profesorem Richard E. Patissem, určeného pro výuku začátečníků v programování. Princip ovšem není vázán na programovací jazyk, takže existují varianty i v jiných programovacích jazycích. Také existují i lokalizované varianty, především v češtině, díky původu slova robot a také zásluhou rozsáhlé komunity. [31, 32]

Obrázek 7 Úkol Robot_Karel – úvodní menu

Karel je robot, který se pohybuje a interaguje s předměty na dvourozměrném poli, kde se snaží na základě svého programu řešit úkoly. Pole obsahuje zdi, které blokují Karlův pohyb, značky, které může Karel sbírat a pokládat, a domeček, který případně značí Karlův cíl. Karel má také několik smyslů a schopností, které mu pomáhají s orientací v prostoru a interakcí s prostorem. Může se dívat do tří směrů (dopředu, doleva, doprava), aby zjistil, jestli se v daném směru nachází zeď, dokáže detekovat, jestli se na poli, na kterém stojí, nachází značka, dokáže poznat směr, ve kterém je otočen, a nakonec dokáže manipulovat se značkami a umístit je buď do svého batohu, pokud je aktivní, anebo je z něho vytáhnou a položit na pole, na kterém stojí. [33]



Obrázek 8 Úkol Robot Karel - prostředí

Cílem tohoto úkolu je doplnit funkce sloužící pro ovládání Karla, inicializaci prostředí a případně i zbylé algoritmy pro průchod plochy a plnění daného úkolu.

4.2.11.1 KarelAlgor.c

Soubor obsahující algoritmy, které pomocí studentem implementovaných příkazů pro pohyb a další akce ovládají Robota Karla, který podle zvoleného algoritmu prochází herní pole a na základě zvoleného „módu“ provádí dané akce. V základu jsou implementovány čtyři algoritmy, z toho tři jsou již hotové a poslední je určen pro doplnění studentem.

- **KarelAlgor_Flood** – Prvním algoritmem je tzv. záplavový algoritmus. Ten funguje tak, že Karel prochází bludiště, přičemž si značí pole, na kterých se již nacházel, pomocí značek. V momentě, kdy projde celé bludiště neboli nenajde místo, na kterém by nebyla značka, vrací se na startovní pozici, na které se nachází Karlův domeček.
- **KarelAlgor_WallTracking** – Dalším je algoritmus, který při průchodu bludištěm sleduje zeď, tedy jednoduše řečeno na křižovatkách zahýbá vždy doleva. Tím teoreticky projde celé bludiště a vrátí se zpět do svého domečku. Toto ovšem platí za předpokladu, že bludiště nemá cesty, na kterých by mohl Robot Karel pořád kroužit kolem jedné zdi.
- **KarelAlgor_Pascal** – Posledním již implementovaným je algoritmus na tvorbu Pascalova trojúhelníku. Tento trojúhelník je složen z čísel, která jsou vždy součtem

čísel, které jsou o řádek výše. Pokud tedy číslo má nad sebou čísla 1 a 3, jeho hodnota bude 4. Robot Karel používá pro zobrazení tohoto obrazce prázdné hrací pole a pro reprezentaci čísel značky, kdy množství značek na jednom poli značí hodnotu v trojúhelníku.

- **KarelAlgor_Collect** – Algoritmem, který má student implementovat, je algoritmus pro sběr značek. Na herním poli beze zdí jsou náhodně rozmístěny značky, které musí Karel sesbírat a následně donést do svého domečku, který je umístěn v protějším rohu, než je Robot Karel. Student musí vytvořit algoritmus tak, aby fungoval v případě, že Karel začíná v jakémkoli z rohů, a i v případě, že se uprostřed herního pole nachází zeď, kterou musí Robot Karel obejít.

4.2.11.2 KarelFunc.c

Tento soubor obsahuje funkce, které zajišťují funkcionalitu prostředí, hracího pole a dalších podpůrných funkcí, jako je generování nebo načítání herní plochy a následné vykreslování. Nacházejí se zde funkce na inicializaci, načítání mapy buď z bitmapového souboru, kde jednotlivé hodnoty složky RGB reprezentují prvek na ploše, nebo z textového souboru, u kterého znaky korespondují s výslednou podobou plochy. Dále jsou zde funkce pro zaplnění plochy prvky, vykreslování závěrečného produktu a pro generování bludiště pro určité algoritmy.

- **KarelFunc_InitiateArray** – Cílem je vytvořit dvourozměrné pole se zadanými rozměry, které tvoří samotnou herní plochu. Také se zde provádí nastavení, zda se mají všechny prvky pole vytvořit se zdí nebo s prázdným polem, což slouží k dalšímu zpracovávání pole, jako je vytváření bludiště a plnění plochy objekty. Vytvořené pole je následně uloženo do struktury spolu s dalšími údaji.

4.2.11.3 KarelUser.c

Posledním souborem, do kterého student přidává svou implementaci, je soubor s funkcemi, které slouží k ovládání Robota Karla, jako je pohyb a rotace, kontrola prostředí a interakce s herní plochou. Tyto funkce vycházejí z popisu programovacího jazyku Karel. [34] Dále jsou zde implementované funkce, které nějakým způsobem interagují s uživatelem. Hlavní součástí je interaktivní a „animované“ menu, ve kterém dochází k výběru algoritmu, varianty herní plochy a dalších nastavení.

- **KarelUser_IsFacingDirection** – Funkce, která vrací hodnotu *true* nebo *false* podle toho, jestli je Robot Karel otočený ve směru, který jsme zadali jako vstupní parametr funkce.
- **KarelUser_CheckWall** – Zkontroluje, zda na poli před Robotem Karlem ve směru, ve kterém je otočený, nachází zeď. Návratová hodnota je *true* nebo *false*.
- **KarelUser_CheckHouse** – Vrací *boolean* hodnotu podle toho, zda se Robot Karel nachází na poli, které obsahuje domeček.
- **KarelUser_TurnLeft** – Funkce, která otočí Karla proti směru hodinových ručiček, tedy změní směr, do kterého je otočený, například ze severu na západ. Pro další možné využití lze vytvořit funkci na otočení doprava.
- **KarelUser_Move** – Posune Robota Karla o jedno políčko ve směru, ve kterém je otočen. V této funkci je také potřeba implementovat možnost, že se ve směru pohybu nachází zeď nebo okraj hrací plochy, což znemožní vykonání pohybu.
- **KarelUser_CheckBeeperField** – Kontrola, zda pole, na kterém se Karel nachází, obsahuje alespoň jednu značku.
- **KarelUser_CheckBeeperBackpack** – Funkce kontrolující, zda se v Karlově baťohu nachází alespoň jedna značka. Závislé na tom, zda je možnost baťohu zapnutá.
- **KarelUser_PickBeeper** – Zvedne značku z pole, na kterém se Robot Karel nachází, a pokud je možnost zapnuta, uloží ho do Karlova baťohu. Je potřeba využít i kontroly, zda se na daném poli nachází značka a zda, pokud je baťoh aktivní, je v něm volné místo.
- **KarelUser_PutBeeper** – Položí značku na pole, na kterém se Robot Karel nachází a kterou Robot Karel získá ze svého baťohu, pokud je batoh povolen, a. Je nutné kontrolovat, zda se na daném poli již nenachází maximální možné povolené množství značek a, pokud je aktivní, jestli má batoh nějaké značky na použití.

4.3 Šablony úkolů a vzorová řešení

Každý úkol, který se v tomto kurzu nachází, má vypracované vzorové řešení, které by měly být bez chyb, tedy všechny testy by měly proběhnout bezproblémově, a poskytovat příklad toho, jak se dá daný úkol řešit. Vyučující může využít vzorového řešení pro demonstraci nebo v případě dotazů a problémů studentů s řešením. Toto řešení ovšem nemusí být jediné

správné a pokud student implementuje své jedinečné řešení, které úspěšně splňuje všechny testy, není důvod toto řešení neuznat.

Šablony jsou vytvořeny tak, aby implementace ze vzorových řešení, které danému úkolu odpovídá, bylo možné přímo aplikovat na onu šablonu. Proto se šablony vytváření z vzorových řešení, u kterých se odstraní kód, který má student za úkol vytvořit sám. Po nahrání šablon na webové repozitář samozřejmě nebudou testy jednotlivých funkcí a samotného programu úspěšné. Také je možné, že bez implementace funkcí budou některé testy způsobovat problémy, jako je nesprávné ukončení nebo přímo pád programu při testování, proto musí být tyto eventuality ošetřeny jak v šabloně, tak v odpovídajícím řešení.

Kvůli testování již musí být v šabloně funkce, které mají základní ošetření vstupů a návratových hodnot. To je jednak z důvodu, že jednotkové testování potřebuje pro spuštění a následné testování znát přesný název testované funkce. Pokud bychom při sestavování aplikace z testů volali funkci, která neexistuje, sestavení může skončit chybou. Z tohoto důvodu jsou také ošetřeny vstupy funkce, protože při testování se kontroluje i právě ošetření vstupů a bez něho může dojít i k pádu programu.

Ke každému úkolu se také přidává soubor README, který obsahuje popis úkolu vycházející z již existujících dříve používaných zadáních, a taktéž samotné cíle, potažmo zadání toho, co má student za úkol.

5 TESTOVÁNÍ SPRÁVNOSTI VYPRACOVANÝCH ŘEŠENÍ

Klíčovou součástí inovace kurzu je implementace automatického vyhodnocování správnosti řešení vypracovaného studenty. Při běžném způsobu kontroly, tedy ručním sestavení a kontrole kódu projektu učitelem, mohou být časové nároky při větším počtu studentů enormní, nemluvě o možném přehlédnutí chyb nebo ulehčení neodpovídajících požadovanému řešení ze strany studenta. Díky využití jednotkových testů lze tento problém vyřešit, neboť student může okamžitě vidět stav jeho řešení a případně ho může i dodatečně opravit. Zároveň vyučující může pouze zkontrolovat stav odevzdání a jeho správnost. Vždy si také může vypracované řešení zkontrolovat manuálně, buďto procházením výstupu z testování nebo přímo stažením řešení a osobní kontrolou.

Řešení, se kterým přijde student by mělo být proveditelné bez chyb na jakémkoliv z hlavních desktopových operačních systémů, jelikož proces sestavování na serveru provádí sestavení jak pro OS Windows, tak pro Linuxovou distribuci, na kterém se následně provádí další testy a kontroly.

Testování správnosti probíhá na serveru GitLab, student si ovšem může dané testy spustit i lokálně, a tak se přesvědčit o správnosti.

5.1 Provádění jednotkových testů

Jak už bylo zmíněno, testování se provádí hlavně na serveru GitLab, ovšem lze využít i lokální verze jednotkových testů. V prostředí Qt Creator k tomu slouží výběr spuštění v menu *kit selector.* Zde lze přepínat mezi hlavním řešením a řešením s jednotkovými testy.

V případě spouštění na serveru jsou jednotkové testy jedním z kroků CI, kdy po spuštění a sestavení projektu dochází k jejich provedení a vyhodnocení. Výsledky testů si může student zobrazit a případně zjistit, kde nastala při jednotkovém testování chyba.

5.2 Testování výstupu programu

Kromě jednotkových testů je v příkladech testování výstupu programu prováděno pomocí dvou různých způsobů. Pomocí porovnávání výstupu studentského řešení se vzorovým a zároveň kontrola práce s pamětí. Porovnávání výstupu již dříve existovalo jako kontrola v předmětu ADS, ovšem bez následné kontroly práce s pamětí.

5.2.1 Ověřování integrity paměti

Pro tento způsob testování slouží program Valgrind, konkrétně nástroj Memcheck. Toto testování je v našem případě primárně určeno pro běh na serveru při závěrečné kontrole, ovšem lze jej provádět i lokálně. Program však v době psaní této práce není dostupný pro operační systém Windows bez instalace dalších podpůrných programů a složitého nastavování. Pro tuto kontrolu je tedy doporučeno využít systému GitLab pro závěrečnou kontrolu, ve kterém lze odevzdané řešení dále měnit, nebo počítače se systémem Linux nebo macOS.

Cílem této kontroly je otestovat, zda student v příkladech, ve kterých pracuje s pamětí, dokáže správně paměť alokovat, realokovat nebo uvolnit, a jestli dokáže identifikovat místa, kde se s vlastní alokací paměti pracuje.

5.2.2 Porovnávání výstupu

Další způsob testování výstupu je porovnávání výstupu programu, tedy výpisu z konzole z programu po poskytnutí vzorových vstupů, a vzorového výstupu, který byl vytvořen pomocí vzorového řešení. Po otestování studentského řešení se vytváří soubor *.diff*, který zobrazuje rozdíly mezi výstupy.

Tento postup má své výhody, ale i nevýhody. Výhodou je, že kontrola je přesnější a poskytuje menší prostor pro chybu nebo možné „ošizení“ řešení jednodušším způsobem. Kontrolování tímto způsobem může ovšem mít nežádoucí vliv na obtížnost úkolů, kdy student musí výstup vytvořit tak, aby přesně odpovídal vzoru, včetně formátu a znaků, jako jsou mezery, tabulátory, velká a malá písmena. Toto lze ovšem eliminovat tím, že výstup vytváří funkce, která je již před vytvořená, v našem případě se jedná o funkci *main* v hlavním souboru. Další nevýhodou může být také omezování studenta tím způsobem, že úkol lze řešit pouze určitými způsoby a tím limituje studentovu tvořivost.

6 INTEGRACE S REPOZITÁŘEM GITLAB

Hlavním cílem inovace kurzu bylo přidat každému projektu jednotkové testy, potažmo testovaní obecně. Testy se posléze provádějí na webovém repozitáři GitLab, který zároveň slouží jako distribuce zadání jednotlivých úkolů. Pro získání a následné odevzdání úkolů lze využít systému Git, který pomocí jednoduchých příkazů nebo případného grafického prostředí umožnuje vytvoření lokálního repozitáře se zadáním, který student otevře ve vývojovém prostředí, provádí v něm změny a následně pro kontrolu nahraje zpět na webový repozitář.

6.1 Získávání šablon úkolů

Všechny úkoly, které bude student během kurzu řešit, jsou nahrány na webovém repozitáři GitLab. Zároveň jsou všechny úkoly dostupné od samotného začátku, proto studenti, kteří už mají pokročilejší znalosti, nemusí čekat, až bude daný úkol zadán, ale mohou pracovat dle svého tempa.

Pro stažení pomocí verzovacího systému Git se využijí příkaz *git clone* a adresy na projekt v repozitáři. Ten se liší podle toho, zda student má přidané nastavení pro SSH zabezpečení nebo jestli využije přímo HTTPS adresu projektu, u které se musí navíc přihlašovat pro získání daného projektu.

Samozřejmě student může úkol stáhnout i v archivačním formátu .zip nebo .tar, potom ovšem nedojde k vytvoření souborů pro systém Git a student přichází o jednodušší stahování šablon, a hlavně o jednoduché nahrávání vytvořených řešení.

6.2 Nahrávání studentských řešení

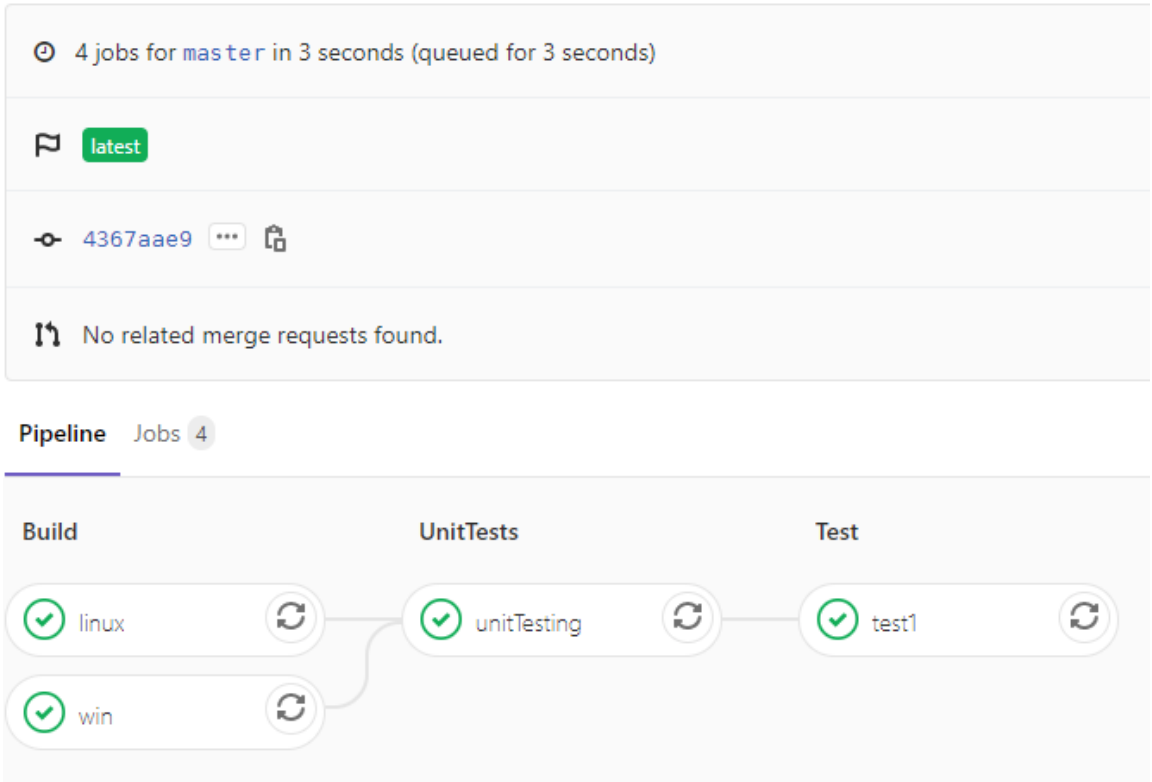
Podle zvolené varianty získání zadání úkolu může student pro nahrání využít systému Git. Pomocí příkazu *git add .* se přidají všechny změny v lokálním repozitáři do tzv. pracovní oblasti, příkazem *git commit -m „message“* se uloží změny a přidá zpráva o provedených úpravách a nakonec se příkazem *git push* nahrají změny na repozitář. Po zadání těchto příkazů se studentem vytvořené řešení nahraje na repozitář a automaticky provede kontrola.

Nahrávat řešení lze také přímo editací a přepsáním souborů na GitLabu pomocí integrovaného textového editoru, nebo také přímo nahrazením dotyčného souboru. Tyto možnosti ovšem nejsou doporučované, protože můžou být zbytečně zdlouhavé a také náchylné na chyby.

6.3 Provedení automatické kontroly řešení

Provádění kontroly a vyhodnocení studentem vytvořeného řešení jsou prováděny pomocí nástroje CI/CD, sloužícího pro tzv. nepřetržitou integraci a distribuci nebo dodání. Tento nástroj se spouští pokaždé, když na daném repozitáři dojde k nějaké změně, například k přidání nového kódu nebo oprava chyby.

V prvním kroku se projekt sestaví, a to na operačním systému Windows a Linux. Cílem je vytvoření spustitelného souboru, na kterém je ověřena spustitelnost kódu na obou platformách. Na verzi vytvořené pro Linux budou následně prováděny další testy. Dalším krokem je provedení jednotkových testů. Těch může být i více, v závislosti na počtu testovaných vstupů a rozsahu úkolu. Výsledkem testů je soubor s výstupem, jehož obsah je zkontrolován, zdali obsahuje informaci o tom, že byly testy provedeny bez chyb. Tento a další krok lze nastavit tak, aby s případným nesplněným některé z částí testu bylo zacházeno jako s chybou, kdy se další testy nebudou provádět, nebo jako varování, kdy se další testy i přes nějakou chybu provedou. Posledním krokem je porovnávání výstupu, kdy samotný program prochází kontrolou paměti pomocí programu Valgrind a porovnávání vytvořeného výstupu se vzorovým výstupem.



Obrázek 9 Automatické testování řešení po nahrání nástrojem CI/CD

7 TESTOVÁNÍ V REÁLNÉ VÝUCE

Jednou se zásad pro vypracování této bakalářské práce je otestování vytvořených úkolů v reálné výuce. Během procesu vývoje docházelo k průběžnému testování vytvořených úkolů, ale kvůli malému počtu testujících a možné „zaslepenosti“ tvůrce nelze odstranit všechny možné chyby. Nasazení příkladů do ostré výuky proto poskytuje ideální prostředí pro testování a odladění.

7.1 Kontext

Předmět Programovací metody je určen pro studijní program Aplikovaná informatika v průmyslové automatizaci ve specializaci Inteligentní systémy s roboty. Předmět je vyučován v prvním ročníku v zimním semestru.

Existují další podobné předměty, Algoritmy a datové struktury a Programování v jazyce C/C++, které jsou zaměřené na výuku programování v jazyce C a C++, přičemž je jejich styl příkladů a testování stejný jako u této práce, tedy využívají vytvořené šablony úkolů a kontrolu jejich řešení pomocí systému GitLab.

7.2 Průběh testování

První zkouška probíhala ke konci zimního semestru 2019. Testování muselo proběhnout až ke konci semestru z důvodu, že předmět se vyučuje již v prvním ročníku a mnoho studentů se s jazykem C, případně programování obecně, ještě nesetkalo. Proto většinu semestru byly studentům objasňovány samotné základy, aby byly vůbec schopni s vytvořenými úkoly pracovat a porozumět jim. Taktéž kvůli časové vytíženosti bylo možné vyzkoušet příklady pouze na jedné skupině, i když s více návštěvami během pár týdnů.

Studenti dostali zadání k příkladu Math_library a měly za úkol ho vyřešit. Během tvorby řešení byl u studentů kontrolován postup a následně správnost řešení. Díky integraci jednotkových testů přímo v úkolech nebyla řešení odesílána na finální kontrolu na webový repozitář, bylo tedy pouze ověřeno, že studenti jsou schopni příklad správně vypracovat.

Další testování probíhalo nepřímo v předmětech ADS a PJC, které oba probíhaly v následujícím letním semestru v oboru Softwarové inženýrství. Zde opět nebylo testování ideální, protože díky krizi s COVID-19 byla většina semestru řešena formou distanční výuky.

7.3 Poznatky a úpravy

Během testování jak kurzu Programovací metody, tak předmětů ADS a PJC vznikly poznatky a podněty pro úpravu jak ze strany studentů, tak i od vyučujících.

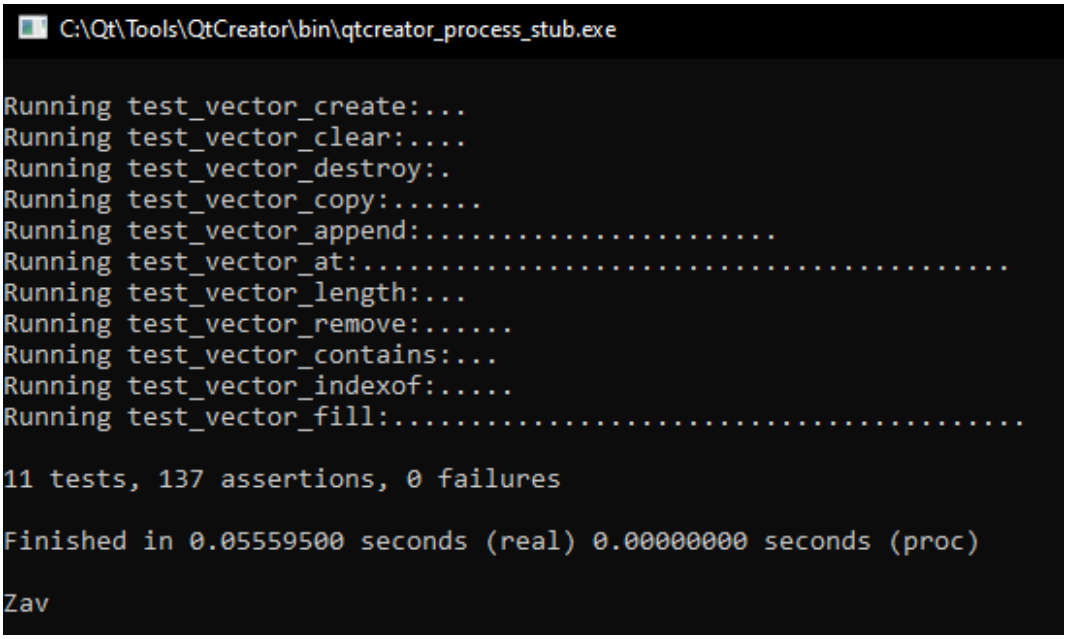
7.3.1 Poznatky z předmětu PME

Hlavním poznatkem je problém s celkovou koncepcí předmětu. Jelikož je předmět pro většinu studentů první zkušenost s programováním v jazyce C, je nutné je seznámit se základy, aby vůbec byly schopni s úkoly pracovat. Předmět můžeme přirovnat ke kombinaci předmětů ADS a PJC v oboru Softwarové inženýrství. V tomto oboru se navíc předpokládá, že studenti již mají nějaké zkušenosti s programováním buď přímo v jazyce C nebo jiném jazyce. Nejlepším způsobem řešení by bylo přidání předmětu Základy jazyka C i pro obor Inteligentní systémy s roboty a posunutí předmětu Programovací metody do dalšího semestru, čímž by bylo možné seznámit studenty se základy a následně projít i příklady vytvořené v této práci.

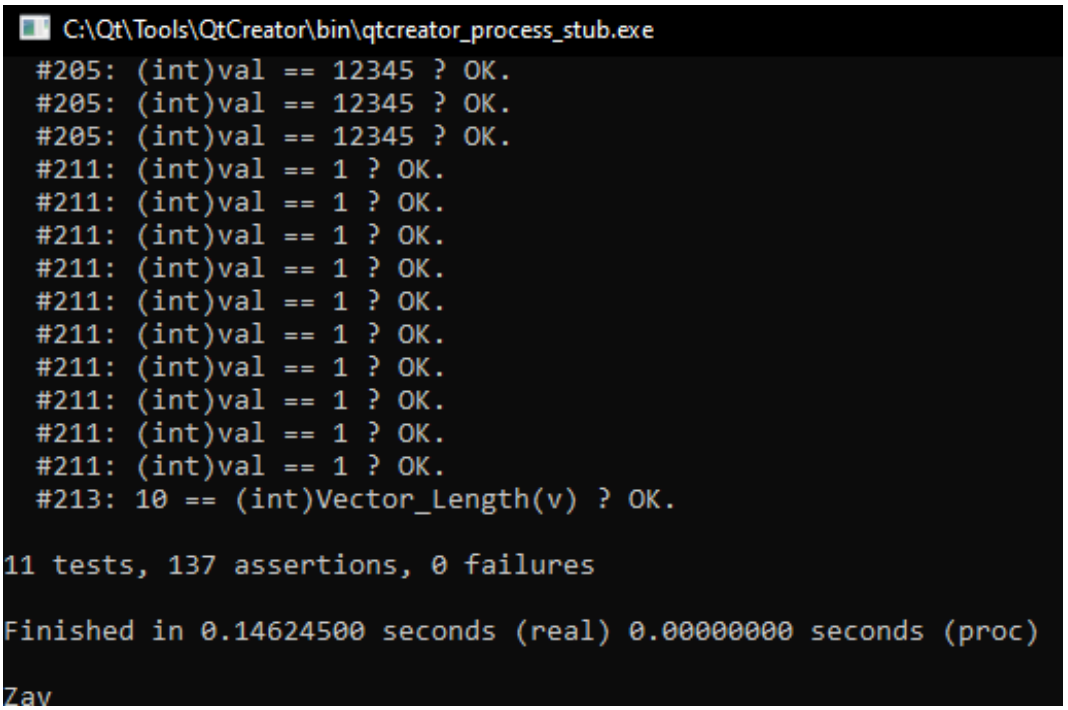
7.3.2 Poznatky z dalších předmětů

Následující poznatky vznikly na základě podnětů pro kurzy v letním semestru, hlavně z předmětu ADS. Množství podnětů zde bylo větší, hlavně díky většímu testovanému vzorku a kvůli nasazení úkolů již na začátku semestru. Taktéž zde není nutné studenty seznamovat s programovacím jazykem C, protože již absolvovali kurz, kde se seznámily se základy jazyka.

Jedním z podnětů byl ohledně knihovny sloužící pro testování, MinUnit. Hlavním problémem byla nepřehlednost toho, co se momentálně testuje. Zobrazení výsledku daného testování probíhalo pouze formou vypsání tečky. Pro zlepšení přehlednosti lze knihovnu upravit tak, že při spuštění testů je definována úroveň testování, tedy co všechno se má při testování uživateli zobrazit. Úrovní je několik, například pro vypsání názvu testu a následné potvrzení jeho úspěšného splnění nebo vypisování při každém volání funkcí *mu_assert*. Kromě popisů při samotném testování bylo doporučeno přidat popisky i k daným funkcím s logikou testování, aby bylo jasné, co se v dané funkci testuje a proč.



Obrázek 10 Varianta MinUnit se zobrazením testovaných funkcí



Obrázek 11 Varianta MinUnit se zobrazením všech testovacích úkonů

Dalším častým problémem byla práce s pamětí neboli přiřazování, realokování a uvolňování. Setkat se s ním můžeme hlavně u složitějších úkolů, jako je například Vector, který patří mezi složitější příklady a je na tuto problematiku zaměřen. Bohužel kvůli charakteru jazyka C nelze práce s pamětí zjednodušit více. Studenti využívají dodanou knihovnu myMalloc, která zjednodušuje práci s pamětí tím, že zobrazuje, kdy a kolik paměti se přiřazuje jakému prostředku. Studenti na OS založeném na UNIXu mohou využít také programu Valgrind pro kontrolu prostředků paměti.

8 MOŽNÁ VYLEPŠENÍ A ÚPRAVY

Během tvorby úkolů vzniklo několik potencionálních problémů a také části, které by bylo možné vylepšit. Některé vylepšení by bylo možné implementovat lehce, u některých by ovšem bylo nutné předělat větší část úkolů, aby dané vylepšení fungovalo. Tyto návrhy nejsou založeny na studentských podnětech.

8.1 Studentem vytvořené jednotkové testy

V první řadě během tvorby úkolů vznikl návrh na jednotkové testy, které by nebyly již předpřipravené, jako je tomu v momentálním stavu, ale student by si je implementoval sám. Student by tak byl seznámen i se samotnou tvorbou testů, a nejen s jejich plněním. Vyvstává však problém, jak potom bude probíhat kontrola, protože kontrola úkolů probíhá hlavně díky jednotkovým testům a možnost, že by si student mohl testy vytvořit tak, aby v podstatě nic nekontrolovaly, není zrovna žádoucí.

Jednou z možností řešení je vytvoření testů, které budou kontrolovat jednotkové testy. Toto se ovšem může ukázat jako problematické, protože k tomuto jednotkové testy nebyly vytvořeny a musely by být upraveny, případně vytvořeno úplně nové řešení, které by tento způsob testování podporovalo.

Další z možností implementace je tedy vytvoření pouze jednoho úkolu s touto možností, kde by si student tvorbu jednotkových testů vyzkoušel, a to buď úplně bez kontroly, nebo s kontrolou limitovanou, kdy by docházelo ke kontrole pouze například počtu provedených testů.

8.2 Implementace se systémem Moodle

I když dochází k automatické kontrole na webovém repozitáři GitLab, tako kontrola vyhodnotí pouze to, jestli testy byly úspěšné nebo ne. Dochází proto ke kontrole ze strany vyučujícího, který může projít studentovo řešení, například v případě, že studentovo řešení neprošlo přes všechny testy, a na základě odevzdaného řešení udělí známku do systému Moodle. Tento proces by šel teoreticky taktéž automatizovat, protože repozitář GitLab i systém Moodle podporuje tvorbu vlastních pluginů, a student se do obou systémů přihlašuje stejnými údaji, šlo tyto systémy propojit tak, že by při uplynutí termínu odevzdání došlo k vyhodnocení odevzdaného řešení, například podle počtu úspěšných testů, a připsání známky danému studentovi v daném kurzu.

Tento návrh by ovšem byl složitější na provedení a ve výsledném řešení by bylo stále jednoduší známkovat manuálně.

8.3 Další příklady

Největším problémem kurzu Programovací metody je ten, že se vyučuje v prvním ročníku, a proto je potřeba některé studenty seznámit s úplnými základy programování, aby vůbec mohli začít s řešením těchto úkolů. Bylo by proto přínosné, aby byly přidány příklady na úplné základy, jako jsou základní operace s proměnnými nebo funkcemi, které je potřeba znát k tomu, aby student pochopil, jak má řešení pro úkoly vytvářet. Nastává ovšem problém, zda do těchto úkolů implementovat jednotkové testy, nebo zda vůbec nějak tyto jednoduché úkoly přidávat a řešit, protože pro správné fungování jednotkových testů je potřeba projekt, který již obsahuje pokročilejší funkce jazyka.

8.4 Inicializace studentských repozitářů

Na začátku semestru dochází k vytváření repozitářů pro studenty, kteří mají předmět zapsaný a jeho naplnění úkoly pomocí externího nástroje, který využívá studentských údajů, které se musí získat exportováním ze systému IS/STAG nebo Moodle. Tento postup je ovšem zdlouhavý a může dojít k chybám, jako je vynechání studenta nebo nahrání špatných verzí úkolů. Možným vylepšením je tedy vytvoření nástroje, který díky případné implementaci jako plugin pro systém Moodle může automaticky získat studenty zapsané v konkrétním kurzu a na GitLab serveru jim automaticky vytvořit repozitář a nahrát do něj specifikované úkoly. Toto navrhované řešení je ovšem jedno ze složitějších navrhovaných, protože je potřeba pracovat s více systémy, kdy každý má jinou strukturu a je napsaný v jiném programovacím jazyce.

ZÁVĚR

Hlavním cílem této práce bylo inovovat úlohy pro kurz Programovací metody. Úkoly byly vytvořeny na základně již existujících, popřípadě mírně upravených zadání z minulých let. Protože úkoly byly zadávány pouze formou textového zadání, vytvořil jsem vzorová řešení, která budou bez chyb procházet kontrolou. Z nich jsem následně vytvořil šablony, do kterých budou studenti doplňovat svá řešení. K těmto úkolům jsem implementoval systém pro jednotkové testování neboli kontrolu správnosti studenty vypracovaného řešení, který již během tvorby, a posléze i jako závěrečná kontrola, testuje vytvořené řešení a oznamuje, kde nastala případná chyba v implementaci. Také jsem implementoval způsob centralizovaného získávání, odevzdávání a kontroly úkolů pomocí webového repozitáře GitLab, se kterým může student pracovat pomocí verzovacího systému Git a webovým rozhraním. Zde kromě kontroly jednotkovými testy probíhá i kontrola práce s pamětí a výstupu programu jako celku. Samotné úkoly jsem následně vyzkoušel v reálné výuce a pomocí poznatků a připomínek získaných od studentů jsem ještě upravil a vylepšil.

V této práci jsem také popsal základní nástroje pro práci s úkoly, případně jejich alternativy, popis programovacího jazyku C a jeho vlastností, a základní poznatky pro vyučování programování. Dále popsány samotné úkoly a jejich strukturu, způsoby testování a možná vylepšení do budoucnosti.

Přesto, že se tato práce zaměřuje na kurz, který je zaměřen na výuku jazyka C, po zkouškách v ostré výuce lze říci, že tento způsob vytváření úkolů a jejich kontrola pomocí jednotkových testů by bylo možné využít i u jiných předmětů zaměřených na programování, které pracují i s jinými jazyky a s různými stupni obtížnosti.

SEZNAM POUŽITÉ LITERATURY

[1] Programovací jazyky. *Stránky k výuce informatiky* [online]. Vlašim: Gymnázium Vlašim, c2020 [cit. 2020-04-02]. Dostupné z: http://www.ivt.mzf.cz/seminar/14-programovaci-jazyky/

[2] Programovací jazyky. *Informace a informatika* [online]. Nové Město nad Metují: Střední průmyslová škola, Odborná škola a Základní škola, Nové Město nad Metují, 2015 [cit. 2020-03-18]. Dostupné z: http://info.spsnome.cz/Programovani/Zaklady-Programovaci-jazyky

[3] Nejlepší programovací jazyk pro začátečníky. *Ivánkův blog* [online]. Irsko: Google Ireland Limited, 2011 [cit. 2020-03-18]. Dostupné z: http://ivankuckir.blogspot.com/2011/09/nejlepsi-programovaci-jazyk.html

[4] Úvod do C. *Nechci slevu zadarmo...* [online]. Praha: ČVUT, c2010-2013 [cit. 2020-04-10]. Dostupné z: http://kmlinux.fjfi.cvut.cz/~fabiadav/cecko/poznamky-k-jazyku-c/uvod-do-c

[5] KERNIGHAN, Brian W. a Dennis M. RITCHIE. *Programovací jazyk C*. Brno: Computer Press, 2006, 286 s. ISBN 802510897X.

[6] *Jazyky C a C++ kompletní průvodce - 2., aktualizované vydání*. Praha: Grada, 2011. ISBN 978-80-247-7417-6.

[7] *CMake Overview* [online]. New York: Kitware [cit. 2020-06-30]. Dostupné z: https://cmake.org/overview/

[8] Introduction | Qbs Manual. *Qt Documentation* [online]. Helsinki: Qt Group, c2018 [cit. 2020-06-30]. Dostupné z: https://doc.qt.io/qbs/overview.html

[9] TURUNEN, Tuukka. Deprecation of Qbs. In: *Qt Blog* [online]. Helsinki: Qt Group, 2018 [cit. 2020-06-23]. Dostupné z: https://www.qt.io/blog/2018/10/29/deprecation-of-qbs

[10] *Qbs GitHub* [online]. San Francisco: GitHub, 2020 [cit. 2020-06-23]. Dostupné z: https://github.com/qbs/qbs

[11] VALKOVIČ, Patrik. Lekce 1 - Kompilace v jazyce C a C++. In: *ITNetwork* [online]. Praha: ITNetwork, 2020, 2017 [cit. 2020-03-20]. Dostupné z: https://www.itnetwork.cz/cplusplus/cecko/pokrocile/kompilace-v-jazyce-c-a-cplusplus

[12] Základní nástroje a pojmy. Jak na projekty v jazyce C [online]. David Martinek, c2017 [cit. 2020-07-10]. Dostupné z: http://jaknaprojekty.davidm.cz/nastroje.html

[13] VALKOVIČ, Patrik. Lekce 2 - Kompilace v jazyce C a C++ pokračování. In: *ITNetwork* [online]. Praha: ITNetwork, 2020, 2017 [cit. 2020-03-20]. Dostupné z:

https://www.itnetwork.cz/cplusplus/cecko/pokrocile/kompilace-v-jazyce-c-a-cplusplus-po-kracovani

[14] Schéma překladu. In: *Jak na projekty v jazyce C* [online]. David Martinek, c2017 [cit. 2020-07-10]. Dostupné z: http://jaknaprojekty.davidm.cz/img/schemaprekladu.svg

[15] *GCC, the GNU Compiler Collection* [online]. Boston: Free Software Foundation, c2020 [cit. 2020-07-01]. Dostupné z: https://gcc.gnu.org/

[16] GCC Mission. GCC, the GNU Compiler Collection [online]. Boston: Free Software Foundation, c2020 [cit. 2020-08-03]. Dostupné z: https://gcc.gnu.org/gccmission.html

[17] MinGW. *MinGW* [online]. New York: Software in the Public Interest, c2020 [cit. 2020-07-12]. Dostupné z: http://www.mingw.org/wiki/MinGW

[18] QT GROUP. Qt Creator 4.12.3 for Windows 64bit [software]. 17. června 2020 [cit. 2020-07-02]. Dostupné z: https://www.qt.io/offline-installers/. Požadavky na systém: Win Vista 64, Win 7 64, Win 8 64, Win 10 64; velikost 294 MB.

[19] Qt Creator Manual. *Qt Documentation* [online]. Helsinki: Qt Group, c2019 [cit. 2020-03-23]. Dostupné z: https://doc.qt.io/qtcreator/index.html

[20] User Interface. *Qt Documentation* [online]. Helsinki: Qt Group, c2019 [cit. 2020-03-23]. Dostupné z: https://doc.qt.io/qtcreator/creator-quick-tour.html

[21] ERAN IFRAN. CodeLite 14.0.0 Windows 64bit [software]. 11. května 2020 [cit. 2020-07-02]. Dostupné z: https://downloads.codelite.org/. Požadavky na systém: Win 7 64, Win 8 64, Win 8.1 64, Win 10 64; velikost 38,2 MB.

[22] ECLIPSE FOUNDATION. Eclipse IDE for C/C++ Developers [software]. 15. černa 2020 [cit. 2020-07-02]. Dostupné z: https://www.eclipse.org/downloads/packages/. Požadavky na systém: Win 7 64, Win 8 64, Win 8.1 64, Win 10 64; velikost 239 MB.

[23] FAIGL, Jan. Úvod do verzovacích systémů (informativní). CourseWare Wiki [online]. Praha: České vysoké učení technické v Praze, 2016 [cit. 2020-07-15]. Dostupné z: https://cw.fel.cvut.cz/old/_media/courses/a0b36pr2/lectures/lecture11-slides.pdf

[24] SOFTWARE FREEDOM CONSERVACY. Git 2.27.0 64-bit Windows [software]. 1. června 2020 [cit. 2020-07-02]. Dostupné z: https://git-scm.com/downloads/. Požadavky na systém: Win 7 64, Win 8 64, Win 8.1 64, Win 10 64; velikost 45,8 MB.

[25] Git - About. *Git* [online]. New York: Software Freedom Conservancy, 2020 [cit. 2020-03-23]. Dostupné z: https://git-scm.com/about/

[26] *SSH (Secure Shell) Home Page* [online]. Helsinki: Communications Security, c2020 [cit. 2020-07-01]. Dostupné z: https://www.ssh.com/ssh/

[27] *GitLab About* [online]. San Francisco: GitLab, [2020] [cit. 2020-07-01]. Dostupné z: https://about.gitlab.com/

[28] Introduction to CI/CD with GitLab. *GitLab Docs* [online]. San Francisco: GitLab, c2020 [cit. 2020-03-24]. Dostupné z: https://docs.gitlab.com/ee/ci/introduction/index.html#introduction-to-cicd-methodologies

[29] GitHub - MinUnit. *GitHub* [online]. San Francisco: GitHub, c2020 [cit. 2020-07-01]. Dostupné z: https://github.com/siu/minunit

[30] Valgrind User Manual. *Valgrind* [online]. Cambridge: Mythic-beast, c2019 [cit. 2020-04-02]. Dostupné z: https://valgrind.org/docs/manual/manual-intro.html#manual-intro.overview

[31] *Karel: Home page* [online]. Teneessee: Middle Teneessee State University, [2015] [cit. 2020-06-26]. Dostupné z: https://www.cs.mtsu.edu/~untch/karel/

[32] JEDLIČKA, Oldrich. Kdo je Karel. *Robot Karel: vývojové prostředí* [online]. Jedlička, [2013] [cit. 2020-06-29]. Dostupné z: http://karel.oldium.net/napoveda.html

[33] Karel the Robot: Fundamentals. *Karek the Robot* [online]. Tennessee: Middle Tennessee State University, [2015] [cit. 2020-06-29]. Dostupné z: http://www.cs.mtsu.edu/~untch/karel/fundamentals.html

[34] Karel programming language documentation. *Petr Kadlec -- homepage* [online]. Kadlec, 2005 [cit. 2020-06-29]. Dostupné z: http://mormegil.wz.cz/prog/karel/prog_doc.htm

SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK

JVM Java Virtual Machine

IDE Integrated Development Environment

GNU GNU ‘s not UNIX

GCC GNU Compiler Collection

SSH Secure Shell

RSA Rivest-Shamir-Adleman

FTP File Fransafer Protocol

CI Continuous integration

CD Continuous delivery/deployment

TDD Test-driven development

WSL Windows subsystem for Linux

CSV Comma-separated values

ADT Abstact data type

RGB Red-green-blue

ADS Algoritmy a datové struktury

PJC Programování v jazyce C/C++

PME Programovací metody

SEZNAM OBRÁZKŮ

SEZNAM PŘÍLOH

PŘÍLOHA P I: CD

Obsah přiloženého CD:

- Bakalářská práce v elektronické podobě
- Zdrojové soubory pro šablony zadání
- Zdrojové soubory pro vzorová řešení (šifrované)