NPRG041 – 2025/26 ZS – Cvičení MS – **Malý úkol C02**

Přepínače

Cílem tohoto úkolu je naprogramovat jednoduchou aplikaci, která dokáže zpracovat všechny argumenty, které jí při spuštění byly předány z příkazové řádky. Zpracováním se myslí detekce očekávaných i neočekávaných přepínačů, správné rozlišení jejich typů, detekce navázaných i samostatných hodnot a vypsání všech takových nalezených přepínačů a hodnot na standardní výstup.

Z pohledu délek názvů přepínačů budeme rozlišovat dvě varianty, a to *krátké* a *dlouhé*. Ty krátké mají ve svém názvu přesně jeden znak a zapisují se s jednou pomlčkou na začátku (např. `-x`). Dlouhé přepínače jsou uvedeny dvěma pomlčkami a jejich název obsahuje alespoň jeden znak (např. `--grayscale`). Všechny názvy jsou pak case sensitive aneb záleží na velikosti písmen.

Bez ohledu na délku názvů budeme rozlišovat další dvě varianty přepínačů. U první z nich nám jde jen o detekci jejich přítomnosti, říkat jim proto budeme *flag options*. U druhé varianty očekáváme, že za přepínačem bude ještě následovat nějaká hodnota, vždy jedna. Říkat jim proto budeme *value options*. Kromě nich se ještě můžeme setkat se samostatnými hodnotami (argument nezačínající na `--` ani `-`), které na žádný hodnotový přepínač navázané nejsou. Těm pak budeme říkat *standalone values*.

Zaměřme se nyní na povolené způsoby zápisu jednotlivých typů přepínačů. U těch krátkých totiž můžeme předat jeden za druhým samostatně (např. `-x -y`), stejně tak je ale možné mít jich hned několik najednou za jedinou úvodní pomlčkou (např. `-xy`). V takovém případě ale máme jistotu, že v takové skupině objevíme jen ty krátké. Jinými slovy dlouhé přepínače seskupovat nejde.

V případě dlouhých hodnotových přepínačů bude navázaná hodnota vždy předána samostatně, tedy v bezprostředně následujícím argumentu (např. `--red 255`). U krátkých hodnotových přepínačů taková možnost existuje také (např. `-r 255`), nadále ale můžeme pracovat i s myšlenkou seskupování. V takovém případě ale ve skupině může být nejvýše jeden hodnotový přepínač, a to výhradně jen jako poslední uvedený. Všechny znaky za ním následující se považují za jeho hodnotu (např. `-xyr255`). Není-li ani jeden takový znak, pak bude hodnota uvedena v dalším argumentu (např. `-xyr 255`).

V obecném případě platí, že navázanou hodnotou může být cokoli. Její obsah tedy neřešíme, i kdyby vypadala jako další přepínač (začínala by třeba na pomlčky). Stejně tak nám je jedno, jestli jde o číslo nebo něco jiného, vše pro nás totiž budou jenom obyčejné řetězce. Vyrovnat se musíme i se situací, kdy z posledního argumentu vyplyne nutnost nalezení navázané hodnoty, ta však ale chybí. Program v takové situaci nesmí spadnout. Jak už vyplývá z předchozího textu, pokud argument nezačíná na pomlčky a není hodnotou navázanou na předchozí hodnotový přepínač, jde o samostatnou hodnotu (např. `-x 255`). Dodejme rovněž, že navázané i samostatné hodnoty mohou být v krajním případě i prázdné řetězce.

Očekávané přepínače a jejich typy jsou předem pevně dány. Jelikož zatím nemáme lepší rozumné prostředky, očekává se, že je opravdu přímo do kódu napevno zadrátujeme. Ovšem jen na jednom místě (případně odděleně pro krátké a dlouhé přepínače, ne však opakovaně). Konkrétně očekáváme flagové přepínače `-x`, `-y`, `--grayscale`, `-t` a `--transparent` a hodnotové přepínače `-r`, `--red`, `-g`, `--green`, `-b`, `--blue`, `-a` a `--alpha`. Pro názvy přepínačů vytvoříme globální pojmenované konstanty.

Argumenty je potřeba zpracovat na jeden průchod, a to beze změny jejich pořadí. Kdykoli detekujeme nějaký přepínač nebo hodnotu, v daný okamžik vypíšeme na standardní výstup textovou zprávu, která o příslušné situaci informuje. Konkrétně očekáváme následující typy zpráv:

- `Flag option <`*name*`> detected`
- `Value option <`*name*`> detected with value <`*value*`>`
- `Value option <`*name*`> detected but its value is missing!`
- `Unknown option <`*name*`> found!`
- `Standalone value detected <`*value*`>`

Tvar zprávy je nutné vždy dodržet, *name* se nahradí za vlastní jméno přepínače bez pomlček, *value* za navázanou nebo samostatnou hodnotu. Lomené závorky `<>` se ponechají beze změny, každý řádek se ukončí pomocí `std::endl`.

Veškerý zdrojový kód je potřeba rozdělit do vhodně navržených modulů, konkrétně můžeme předpokládat soubor `Main.cpp` obsahující výhradně funkci `main` a dále dvojici souborů `Options.cpp` a `Options.h` pro

veškerý výkonný kód týkající se vlastního zpracování argumentů. Cílem hlavičkového souboru je umožnit oddělení deklarací od vlastní implementace. Automaticky předpokládáme, že všechny naše hlavičkové soubory budou mít ochranu proti nechtěnému opakovanému vložení, a to pomocí mechanismu direktiv `#ifndef`, `#define` a `#endif`.

Abychom s předanými argumenty nemuseli pracovat v podobě tradičního pole céčkových řetězců, hned na začátku funkce `main` nejprve provedeme jejich konverzi do uživatelsky přívětivější podoby kontejneru `std::vector` řetězců `std::string`. Toho lze snadno dosáhnout pomocí triku `std::vector<std::string> arguments(argv + 1, argv + argc)`.

Zamýšlený kód dekomponujeme do vhodně navržených funkcí, základní inspiraci pak můžeme najít v následujícím přehledu. Konkrétní funkce, jejich názvy ani rozhraní však předepsané nejsou, je tedy možné se libovolně odchýlit.

- `process_arguments`: provede zpracování všech argumentů aneb bude obsahovat hlavní cyklus přes jednotlivé argumenty a následné větvení pro dlouhé a krátké přepínače a samostatné hodnoty
- `process_short_option`: zpracuje krátký přepínač aneb na základě již popsaného předdefinovaného seznamu očekávaných přepínačů provede větvení na flagové, hodnotové a neznámé přepínače
- `process_long_option`: totéž pro dlouhé přepínače
- `process_flag_option`: zpracuje krátký i dlouhý flagový přepínač
- `process_value_option`: totéž pro hodnotový přepínač
- `process_unknown_option`: zpracuje přepínač neznámého jména
- `process_standalone_value`: zpracuje samostatnou hodnotu

Pokud jde o hlavní cyklus nad jednotlivými argumenty, očekává se použití iterátorů aneb konstrukce `for (auto it = arguments.begin(); it != arguments.end(); ++it) { ... }`. Největší komplikaci bude patrně představovat zpracování hodnotového přepínače, pokud je jeho hodnota uvedena až v dalším argumentu. V takovém případě můžeme v zásadě zvolit dvě možná řešení: buď takový přepínač zpracujeme zpětně až v momentě nalezení jeho hodnoty, nebo si naopak už v okamžiku jeho nalezení dopředně jeho hodnotu zpřístupníme. V prvním případě budeme muset mezi jednotlivými iteracemi přenášet potřebné informace, ve druhém budeme muset manuálně zasahovat do průběhu cyklu. To však je s iterátory snadné.

Za účelem větvení kódu při vlastní detekci jednotlivých očekávaných i neočekávaných krátkých přepínačů využijeme konstrukci `switch` (a to včetně možnosti uvést více `case` návěští sdílejících stejné tělo). U těch dlouhých to samozřejmě nepůjde, tam využijeme klasické `if`, `else if` a `else` podmínky.

Podstatné je, že funkce `process_flag_option` a `process_value_option` opravdu musí být navrženy tak, abychom je mohli snadno použít pro krátké i dlouhé přepínače současně, tedy s jediným rozhraním. U hodnotových přepínačů navíc potřebujeme, aby naše funkce sama o sobě zvládla najít navázanou hodnotu ve stejném i následujícím argumentu, aniž bychom takový úkol přenášeli na volajícího.

Chceme totiž dosáhnout co nejelegantnějšího kódu právě na straně volajícího, např. ve stylu jediného řádku `if (name == "red") { process_value_option(name, …); }`. Pro dosažení tohoto záměru se nám může hodit trik, kdy několik posledních parametrů funkce může mít stanovené výchozí hodnoty, díky čemuž takové parametry při volání uvádět vůbec nemusíme. Uvědomme si také, že každá funkce může mít jen jednu očekávanou návratovou hodnotu. Pokud by nepostačovala pro předání všech potřebných informací, můžeme využít tzv. výstupních parametrů, tedy obyčejných parametrů předaných modifikovatelnou referencí.

Všechny naše dílčí funkce pak musí být navržené tak, abychom pomocí návratové hodnoty typu `bool` byli schopni informovat o úspěšnosti jejich provedení. V případě úspěchu vrátíme hodnotu `true`, jinak `false`. Za problém považujeme situaci, kdy jsme detekovali alespoň jeden neočekávaný přepínač nebo jsme detekovali sice očekávaný hodnotový přepínač, ovšem bez jeho hodnoty. Detekce chybové situace nevede k přerušení programu, zpracování zbývajících argumentů tedy pokračuje dál bez jakékoli změny.

Informaci o případné nalezené chybě musíme umět dostat až do funkce `main`, tam ji použijeme pro vytvoření návratové hodnoty celého programu, tedy `0` v případě úspěchu a konkrétně `1` naopak.

Abychom vyloučili případné pochybnosti ohledně správného pochopení zadaného problému a všech dílčích požadavků, pojďme se podívat na následující příklad běhu programu: vstupní argumenty `-xwrgb --red --c 255 --grayscale -tb` povedou k návratovému kódu `1` a následujícímu očekávanému výstupu:

```
Flag option <x> detected
Unknown option <w> found!
Value option <r> detected with value <gb>
```

```
Value option <red> detected with value <--c>
Standalone value detected <255>
Flag option <grayscale> detected
Flag option <t> detected
Value option <b> detected but its value is missing!
```

Pro účely ladění poznamenejme, že v rámci vývojového prostředí MS Visual Studio můžeme argumenty předávané při spuštění aplikace snadno nastavit v rámci vlastností projektu. Vše najdeme v nastavení *Project* → *Properties* → *Debugging* → *Command Arguments*.

Odevzdejte všechny vytvořené zdrojové soubory (`*.cpp` a `*.h`). Jména výše uvedených souborů nebo funkcí nejsou pevně vyžadována, můžete si je změnit dle libosti. Hlavním cílem úkolu je naučit se pracovat s textovými řetězci, hlavičkovými soubory, předáváním parametrů referencí a také získat základní představu o práci s kontejnerem vektoru a jeho iterátory. Předmětem hodnocení je samozřejmě i kvalita návrhu, platí i obecné požadavky jako správnost nebo nepoužívání neadekvátních konstrukcí nebo knihoven, se kterými jsme se zatím pracovat nenaučili.

Pokud jde o časté chyby a naopak dobrou praxi, nezapomeňte na následující aspekty. Funkce `main` nesmí obsahovat příliš mnoho kódu, natož nízkoúrovňového. Na první pohled by z ní tedy měla být patrná vysokoúrovňová představa o tom, co program dělá.

V hlavičkových souborech ochranu proti opakovanému vložení používejte vždy, protože nikdy nevíte, kdo a jak je v budoucnosti bude používat. Nepoužívejte ovšem `#pragma once`, nejde o standardní direktivu. Standardní knihovny i uživatelské hlavičkové soubory vkládejte jen v nezbytně nutném kontextu, nevkládejte tedy něco, co se nevyužívá. Do hlavičkového souboru obecně není vhodné automaticky dávat deklarace všech funkcí. Užitečné je to jen pro ty, které chceme ostatním nabídnout k využívání, naopak naše čistě interní funkce tam dávat nebudeme. Jejich deklarace pak můžeme (ale také nemusíme) umístit až na začátku céčkového souboru.

Volte vhodné názvy pro všechny soubory, proměnné, funkce i jejich parametry. Při předávání parametrů nezapomínejte na příznak `const`. Pokud to není záměr, vyvarujte se předávání jakýchkoli objektů složitějších než instance primitivních typů (jako `int` apod.) hodnotou. To platí už i pro řetězce `std::string`. Vhodně tedy používejte reference nebo ukazatele. Pokud nepotřebujete opak, preferujte spíše preinkrementaci namísto postinkrementace. Vyvarujte se opakování podobných nebo dokonce stejných fragmentů kódu. Vyvarujte se také přílišného zanořování podmíněných výrazů.

Technické řešení by nikdy nemělo zakrýt logickou podstatu problému, takže například konverzi našich `bool` chybových příznaků na `int` exit kódy musíme provést explicitně. Nelze se tedy spoléhat na skutečnost, že logické hodnoty jsou ve skutečnosti implementovány číselnými hodnotami, shodou okolností přesně těmi, které chceme. To je jen náhoda, mohlo by tomu být úplně jinak. Hodnoty exit kódů jako takových pak definujeme pomocí pojmenovaných konstant. Při sběru dílčích chybových hodnot napříč našimi funkcemi nezapomeňte na to, že je aplikován mechanismus zkráceného vyhodnocování logických výrazů.

Na úplný závěr dodejme, že pokud se rozhodnete rozvinout celý jmenný prostor `std` nebo i jen jeho část pomocí konstrukce `using`, nikdy tak nedělejte v hlavičkovém souboru. Takový krok by totiž už nebylo následně možné zvrátit.