NPRG041 – 2023/24 ZS – Cvičení MS – **Malý úkol C02**

Argumenty I

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`).

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*.

Pozornost je potřeba věnovat i tomu, jakým způsobem mohou být jednotlivé typy přepínačů specifikovány. 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). 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`).

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`.

Potřebný kód dekomponujeme do vhodně navržených funkcí. To minimálně předpokládá vyčlenění funkce na realizaci zpracování všech argumentů. Tu navrhneme tak, že bude očekávat argumenty v podobě kontejneru `std::vector` řetězců `std::string`, ne ukazatele na céčkové řetězce, jak je dostaneme ve funkci `main`. Do této podoby je tedy potřebujeme nejprve konvertovat, toho však lze dosáhnout snadno pomocí triku `std::vector<std::string> arguments(argv + 1, argv + argc)`.

Abychom se naučili pracovat s dalšími konstrukty, je nutné pro iteraci přes jednotlivé argumenty použít mechanismu iterátoru, který nám kontejner vektoru nabízí. Stačí jen použít konstrukci `for (auto it = arguments.begin(); it != arguments.end(); ++it) { ... }`. Za účelem větvení kódu při vlastní detekci jednotlivých očekávaných i neočekávaných krátkých přepínačů je nutné použít konstrukci `switch`. U těch dlouhých to samozřejmě nepůjde, tam využijeme klasické podmínky.

S ohledem na plánované rozšíření naší aplikace v rámci navazujícího úkolu Argumenty II je potřeba, abychom dále vyčlenili oddělené funkce na zpracování jednotlivých variant detekovaných přepínačů a hodnot, jakkoli je aktuálně budeme používat pouze pro již diskutované vypisování informačních hlášek. Konkrétně tedy předpokládejme funkce `process_flag_option`, `process_value_option`, `process_unknown_option` a `process_standalone_value`.

Podstatné je, že první dvě uvedené funkce musí být navrženy tak, abychom je mohli snadno použít pro krátké i dlouhé názvy současně s jediným rozhraním. Speciálně u hodnotových přepínačů to znamená, že naše funkce sama o sobě musí zvládnout navázanou hodnotu najít ve stejném i následujícím argumentu dle situace, aniž bychom takový úkol přenášeli na volajícího. Cílem je tedy dosáhnout co nejelegantnějšího kódu, např. ve stylu `… if (name == "red") { process_value_option(name, …); } …`.

Pro dosažení předchozího 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 samozřejmě může mít jen jednu 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 jejich úspěšném provedení nebo naopak problémech. V prvním případě vrátíme hodnotu `true`, ve druhém `false`. Za problém považujeme situaci, kdy jsme detekovali alespoň jeden neočekávaný přepínač nebo jsme detekovali sice očekávaný hodnotový přepínač, ovšem bez jeho hodnoty.

Informaci o chybě musíme umět dostat až do funkce `main`, tam bychom ji za normálních okolností použili pro vytvoření návratové hodnoty celého programu, tedy `0` v případě úspěchu a např. `1` naopak. Problém ale je, že ReCodEx s takovými kódy neumí pracovat a cokoli jiného než `0` považuje za selhání testů. Z toho důvodu vždy napevno vraťte právě hodnotu `0` bez ohledu na situaci.

Pro účely ladění dodejme, že v rámci vývojového prostředí MS Visual Studio můžete argumenty předávané při spuštění aplikace snadno nastavit v rámci vlastností projektu. Vše najdete 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 nutné dávat deklarace všech funkcí. Užitečné je to jen pro ty, které chceme ostatním nabídnout k využívání, naopak naše čistě interní funkce tam dávat nebudeme.

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.