NPRG041 – 2023/24 ZS – Cvičení MS – **Malý úkol C06**

Aritmetické výrazy I

Tématem další dvojice úkolů budou aritmetické výrazy a práce s nimi. V tomto úkolu se nejprve zaměříme na schopnost reprezentace takových výrazů, jejich vypisování v podobě postfixové a infixové notace a také jejich vyhodnocování. V navazujícím úkolu bychom se následně věnovali jejich parsování ze vstupních řetězců zapsaných v infixové notaci.

Konkrétně budeme předpokládat jednoduché aritmetické výrazy nad celočíselnými hodnotami. Přesněji řečeno, jedinými vstupními hodnotami (aneb jednoduchými operandy, tedy faktory) mohou být nezáporná celá čísla (takže přirozená čísla nebo nula, a to bez uvedení unárního znaménka). Při vyhodnocování však už mezivýsledky jednotlivých operací mohou být i záporné, vždy však celočíselné.

Z operací konkrétně povolíme jen sčítání, odčítání, násobení a dělení. Používat pro ně budeme v uvedeném pořadí následující symboly operátorů: `+`, `-`, `*` a `/`. Všechny uvedené operace podle obvyklých konvencí vnímáme jako binární zleva asociativní operace. To znamená, že např. výraz `3+5+7-9` vnímáme implicitně uzávorkovaný jako `((3+5)+7)-9`.

Použití pomocných kulatých závorek `(` a `)` může být nezbytné pro vynucení konkrétně zamýšleného pořadí vyhodnocování jednotlivých operací, pokud by nám výchozí priority operátorů nevyhovovaly. To nastává např. u výrazu `3*(7-4)`. V tomto smyslu opět předpokládáme obvyklé chování, tedy že operace násobení a dělení mají vyšší prioritu (precedenci) než operace sčítání a odčítání. Podstatné pro nás v rámci tohoto úkolu je, že závorky samotné při reprezentaci výrazů nijak uvažovat nebudeme aneb postačí nám pouze správně podchytit vnitřní stromovou strukturu výrazů. Z ní už případné uzávorkování vyplyne samo.

Pro zmíněnou reprezentaci využijeme koncept dědičnosti a navrhneme následující hierarchii tříd, pomocí kterých budeme schopni reprezentovat jednotlivé uzly stromu, od kořenového, přes vnitřní, až po listové. Konkrétně budeme předpokládat abstraktní třídu `Node` jakožto společného předka všech uzlů, odvozenou finální třídu `NumberNode` pro číselné faktory (listové operandy), abstraktní odvozenou třídu `OperationNode` pro společnou reprezentaci všech našich binárních operací a z ní odvozené finální třídy pro jednotlivé operace, tedy `AdditionNode`, `SubtractionNode`, `MultiplicationNode` a `DivisionNode` pro operace sčítání, odčítání, násobení a dělení v tomto pořadí.

Pro třídu listové číselné hodnoty se očekává implementace parametrického konstruktoru `NumberNode(int number)`. Pro jednotlivé uzly operací to pak bude konstruktor `OperationNode(Node* left, Node* right)`. Prostřednictvím obou parametrů `left` a `right` dostaneme platné ukazatele na korektně vytvořené uzly reprezentující podstromy levého resp. pravého operandu dané operace. Očekáváme, že oba tyto uzly byly vytvořeny pomocí dynamické alokace, předané ukazatele si tedy uložíme a stáváme se vlastníky takových uzlů. To znamená, že přebíráme plnou odpovědnost za jejich následnou dealokaci. Jinými slovy je nutné kromě uvedených konstruktorů také korektně implementovat příslušné destruktory.

Názvy uvedených tříd uzlů a rozhraní jejich konstruktorů je nutné dodržet, všechny ostatní jejich metody je možné si navrhnout dle svého uvážení. V návaznosti na další požadavky by se nám však z praktických důvodů mohlo hodit, pokud bychom navrhli enumerační třídu se dvěma různými hodnotami (např. `NUMBER` a `OPERATION`), pomocí kterých bychom pak byli snadno schopni rozlišit listové číselné uzly od uzlů operací.

Při návrhu potřebných metod naplno využijeme mechanismů dědičnosti aneb hlavně virtuálních funkcí. Cílem je dosáhnout optimální implementace ve smyslu neopakování podobných nebo dokonce stejných fragmentů kódu. Stejně tak se budeme snažit v jednotlivých uzlech ukládat jen opravdu nezbytně nutné informace, ty odvoditelné nikoli.

Pro reprezentaci jakkoli komplikovaného výrazu pochopitelně postačí, když budeme v ruce držet ukazatel na jeho kořenový uzel. Z praktického pohledu na věc by ale bylo nešťastné, pokud bychom uživatele nutili pracovat s výrazy jako celky právě v takovéto podobě tradičních ne zrovna přívětivých ukazatelů. Zvláště když ještě pracujeme s dynamickou alokací. Jinými slovy naši stromovou strukturu budeme považovat jen za interní záležitost, od které chceme koncové uživatele pokud možno odstínit, stejně jako jim chceme nabídnout rozhraní, které bude dostatečně elegantní.

Z tohoto důvodu navrhneme třídu `Expression`. Jejím cílem bude právě zapouzdření jednoho konkrétního aritmetického výrazu. Konstruktor nabídneme jediný, a to `Expression(Node* root)`. Opět předpokládáme, že dostaneme ukazatel na kořenový uzel vytvořený dynamickou alokací, a tedy přejímáme odpovědnost za jeho budoucí dealokaci (včetně celého případného podstromu), a tedy musíme dodat i odpovídající destruktor.

U třídy výrazu dále nabídneme členskou funkci `void print_postfix(std::ostream& os = std::cout) const`, prostřednictvím které budeme schopni do předaného výstupního streamu vypsat serializaci našeho výrazu v postfixové (reverzní polské) notaci. Jelikož se v ní může vyskytovat více čísel hned za sebou, budeme vždy mezi čísly i operátory důsledně vypisovat jednu oddělovací mezeru. Pro ilustraci se podívejme na konkrétní příklad: pro výraz `1*2+3*(4+5)-6` v infixové notaci vypíšeme `1 2 * 3 4 5 + * + 6 -` v té postfixové.

Analogicky nabídneme i členskou funkci `void print_infix(std::ostream& os = std::cout) const` pro vypsání výrazu v infixové notaci. Tentokráte kolem čísel, operátorů ani závorek žádné mezery vypisovat nebudeme, nebude to totiž potřeba. Pokud jde o závorky, ty jsou v tomto případě na rozdíl od postfixové notace nezbytné. Budeme je však vypisovat pouze v nutných situacích ve smyslu věrného zachování zamýšlené vnitřní struktury výrazu a jeho následného korektního vyhodnocení.

To znamená, že při serializaci uzlů našich operací použijeme tato dvě pravidla: levý operand obalíme závorkami právě tehdy, když reprezentuje podstrom operace, a to operace s nižší precedencí, než je naše; pravý operand reprezentující uzel operace obalíme závorkami vždy kromě případu, kdy má naopak vyšší precedenci, než máme my. Pokud bychom měli vstupní výraz např. `(7+(9-(3*1))/3)-(5-1)`, vypíšeme jej jako `7+(9-3*1)/3-(5-1)`. Právě za účelem detekce jednotlivých situací se nám na tomto místě bude hodit schopnost snadno rozlišit uzly čísel od uzlů operací. Potřebovat také budeme přetypování, a to pomocí např. tradiční konstrukce `(OperationNode*)p`, kde `p` je ukazatel na obecný `Node`.

Úplně poslední metodou třídy `Expression` bude `int evaluate() const`. Jejím cílem bude provést vyhodnocení celého aritmetického výrazu aneb spočítání jeho výsledné hodnoty. Ta bude předána pomocí návratové hodnoty. Připomeňme, že všechny operace (tedy zejména dělení) vnímáme jako celočíselné. Případné přetečení nedetekujeme a ani nijak neřešíme. V rámci tohoto úkolu zatím neošetříme ani případné dělení nulou.

Odevzdejte všechny vytvořené zdrojové soubory (`*.cpp` a `*.h`) kromě hlavního souboru `Main.cpp` s funkcí `main`. Ten totiž opět již je součástí připraveného testu. Předpokládejte, že obsahuje direktivu `#include "Expression.h"`.

Hlavním cílem úkolu je prokázání schopnosti návrhu složitější hierarchie tříd s dědičností a virtuálními funkcemi, a to včetně práce s finálními a abstraktními třídami, čistě virtuálními metodami, konstruktory a destruktory. Dalším hlavním cílem je práce s dynamickou alokací u jednotlivě alokovaných objektů s netriviálním životním cyklem. Dále jde o práci s ukazateli, použití enumerační třídy a také aplikaci obecné myšlenky jakési wrapper třídy pro zapouzdření jiných struktur s cílem nabídnutí elegantnějšího rozhraní a zakrytí interních, technických a dalších implementačních detailů.

Opět se očekává dodržení obvyklých požadavků na naše úkoly. Pokud jde o specifické požadavky, všechny naše operace, jejich operátory i precedence můžeme v kódu zadrátovat napevno, a to i na více místech. Náš kód je totiž svázaný jen s našimi aritmetickými výrazy a pro jiné situace (typy výrazů) přímo aplikovatelný stejně nebude. Konstanty pro symboly operátorů, závorek a mezer však i tak zadefinujeme pomocí globálních pojmenovaných konstant. To se týká i případných konstant pro jednotlivé úrovně precedencí, pokud se je rozhodneme využít a neporovnávat operátory přímo.

V rámci uzlů našeho stromu budeme uchovávat potřebné informace a nabízet odpovídající funkcionalitu právě v těch uzlech, kterých se mají logicky týkat. To se zejména týká abstraktního uzlu `OperationNode`, který má zapouzdřit vše, co je společné pro všechny naše konkrétní uvažované operace. Naopak řešit vyhodnocování jednotlivých operací na úrovni tohoto uzlu pomocí konstrukce `switch` (nebo jinak analogicky) by bylo nevhodné a proti principům dědičnosti.

Vypsání výrazů v postfixové resp. infixové notaci dosáhneme pomocí realizace odpovídajícího průchodu stromem, tedy postorder resp. inorder průchodem doleva do hloubky. Samotný průchod stromem však bude fakticky vykonáván implicitně aneb rekurzivně přímo jednotlivými uzly jako takovými, nikoli nějakým algoritmem, který by s celým stromem výrazu pracoval externě a jako celkem. To by v našem případě opět bylo proti smyslu dědičnosti.

Vyvarujte se opakovaného psaní totožně vypadajících konstruktorů pro uzly konkrétních operací aneb použijte konstrukci `using`. V rámci celé hierarchie musíme pracovat s konceptem virtuálních destruktorů, jinak bychom nebyli schopni dosáhnout korektního chování. Pokud jde o dealokaci našich dynamicky vytvářených uzlů, musí být vždy řešena na místech, které tomu logicky přísluší, tedy právě v již diskutovaných destruktorech, v žádném případě ne jinde.

Pro jistotu ještě dodejme, že dealokaci každého uzlu je nutné vyvolat právě jednou, tedy na ni nesmíme zapomenout, stejně jako ji nesmíme omylem vyvolat vícekrát. S ohledem na dobrou programátorskou praxi to znamená, že dodržíme postup, kdy před dealokací nejprve otestujeme, že příslušný ukazatel není nulovým,

a naopak po provedené dealokaci jej záměrně explicitně vynulujeme.

Jde o preventivní mechanismus, který nám má pomoci bránit se před našimi vlastními chybami v kódu, protože při jakémkoli následném neoprávněném pokusu o použití takového již neplatného ukazatele dojde k okamžitému pádu programu. Pro práci s nulovými ukazateli jako takovými budeme používat výhradně jen speciální literál `nullptr`, nikoli starší přístupy založené na makru `NULL` nebo snad `0` jako takové.