HÁZI FELADAT

Programozás alapjai 2.

Végleges

Németh Dániel
FTYYJR

2019. március 31.

TARTALOM

1. Feladat

```
Szoftver laboratórium II. házi feladat
Teszt Elek (ELEK07) részére:


Kifejezés fa
Készítsen kifejezés fát reprezentáló osztályt! Definiáljon egyszerű
általánosított műveleteket, amelyeket kiértékel a kifejezésfa segítségével!
Valósítsa meg az összes értelmes műveletet operátor átdefiniálással
(overload), de nem kell ragaszkodni az összes operátor átdefiniálásához!
Demonstrálja a működést külön modulként fordított tesztprogrammal! A
megoldáshoz ne használjon STL tárolót!
```

2. Pontosított feladatspecifikáció

A feladat egy kifejezésfát kezelő osztály elkészítése. Az osztály segítségével kifejezéseket hozhatunk létre, tárolhatunk, kombinálhatunk és kiértékelhetünk. Az osztállyal bármekkora, a memóriában elférő kifejezést előállíthatunk (dinamikus memóriakezelés).

Az Expression osztály képes többféle típusú kifejezést is kezelni (int, double) bemenetként és kimenetként, ezt template segítségével fogom megoldani.

Az alábbi funkciókat tervezem megvalósítani:

Expression(): számértéknél nullára inicializál, esetlegesen egyéb használt osztálynál az osztály default konstruktorát hívja.
Expression(érték): a felhasználó által megadott értékre inicializál, ennek az értéknek a típusával fog dolgozni a fa.
Expression(string): Képes infix formátumban megadott stringből kifejezésfát építeni
Támogatott műveletek: +, -, *, /, ^ (hatványozás): két kifejezést ezekkel tudunk összekapcsolni akár helyben is(működik pl a +=, -=, stb. operátorok), akár új változóba (copy constructor)
Kiírás: az osztály képes olvasható formában megjeleníteni a gráfot (postfix, infix vagy prefix alakban, string formátumban). Inserter operátor (<<) alapértelmezetten infixet használ.
Eval(): a kifejezés értékét adja vissza.
Tesztelés: különböző kifejezéseket állítok elő, és a kiértékeléskor kapott értékeket összevetem a c++ beépített kiértékelője által visszaadott értékekkel.

3. Terv

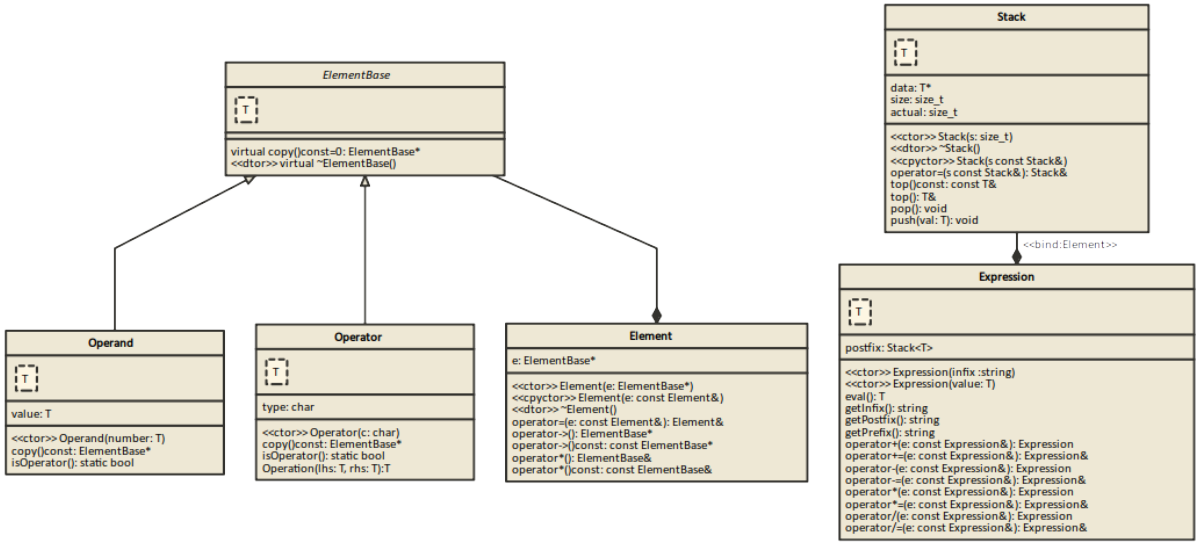
3.1. Objektumterv

A kifejezést egy stackben fogom tárolni postfix alakban, ahol minden elem egy érték vagy egy operátor (heterogán kollekció). Ez előnyösebb egy fával szemben olyan szempontból,

hogy nem kell feltétlenül minden új elem beillesztésénél memóriafoglalást végezni. Bevett algoritmusok állnak rendelkezésre a postfixből és a postfixbe konvertáláshoz.

Az ElementBase absztrakt ősosztályból származnak le az Operator és az Operand osztályok. Az Element osztály becsomagolja az ElementBase típusúakat és érték szerint másolhatóvá teszi. Az Expression osztály egy Element osztályú elemeket tartalmazó stackben tárolja a kifejezést. A stack ezenfelül szükséges még a konverziós algoritmusokhoz.

Globális függvényként elkészítem az Expression-höz tartozó inserter(<<) operátort.



3.2. Algoritmusok

3.2.1. Infixből postfix

Balról jobbra olvas(amíg a string végére érünk):
Ha operandus: push postfixbe.
Egyébként:
Ha nagyobb a precedencia, mint a stakben levőé, vagy a stack üres, vagy nyitó zárójel van benne: push postfixbe
Egyébként: pop minden operátorra, aminek a precedenciája nagyobb vagy egyenlő a beolvasottnál. Beolvasott push stackbe
(: push
): amíg nem) „(”: pop és push postfixbe

3.2.2. Postfixből infix

Létrehozunk egy temp stacket
Balról jobba:
Operandus: push tempbe
Egyébként:
stackből pop az utolsó két elemre
string= operand1 operator operand2
stringet pusholjuk a tempbe
Ha csak egy érték maradt a tempben: ez az infix

3.2.3. Postfixből prefix

Balról jobbra:
Operandus: push stackbe
Operator: stackól pop két operandus, összefűzni az operátort a két operandussal, stringet push stackbe

3.2.4. Kifejezések összekapcsolása operátorral

A meglévő postfix végére push(új operátor), majd push(új expression elemei sorban balról jobbra)

3.2.5. Tesztprogram algoritmusai

A lekódolt metódusokat egy alkalmazási példa alapján meghívja a tesztprogram.
A tesztprogram a standard inputról file végéig olvas. Az első beolvasott adat egy teszteset sorszámot jelent. Ezt követően egy megjegyzés lehet az adott sorban. A beolvasott szám dönti el, hogy melyik teszteset fut a megjegyzés pedig az adott tesztesetre vonatkozhat.
Az 1. tesztben minden első sorból egy infix stringet olvasunk be, minden második sorból pedig az elvárt eredményt.
A 2. tesztben az inputról kapjuk az elvárt infix vagy prefix stringet, a program pedig teszteli a konverzió helyességét.
A 3. tesztben a tesztprogramban meghatározott műveleteket végzünk kifejezésekkel, az inputról pedig soronként kapjuk az elvárt eredményt.

4. Megvalósítás

A feladathoz készítettem egy tesztprogramot és a következő generikus osztályokat: Operand, Operator, ElementBase, Element, Expression, Stack.

Az operandus osztályhoz hozzáadtam a következő elemeket: T getval(), ami az operandus értékét adja vissza, hozzáadtam egy stringbe konvertáló operátort, és egy precedencia-függvényt, ami operandusoknál -1 -et ad vissza.

Az operator osztályba a terven felül bekerült még egy stringet paraméterül kapó konstruktor, egy stringgé konvertáló operátor, egy karaktert visszaadó gettype() függvény, ami az operátor típusát adja vissza, valamint egy precedence() függvény, ami az operátor precedenciáját adja vissza.

Az ElementBase osztályba bekerültek még a következő virtuális függvények: isOperator(), Operation(…), getval(), precedence(), gettype(), és stringgé konvertáló operátor.

Az Element osztály változatlan maradt, az Element publikus része szintén.

A teszteseteken módosítottam a program írása során felmerült igényeknek megfelelően.

A tesztprogram a main.ccp fájlban, míg a többi osztály a megfelelő hpp állományban deklarált és definiált, kivéve az Expression osztályt, ahol a definíciók az Expression.tpp állományban vannak. A továbbiakban részletezem a fontosabb interfészeket és algoritmusokat.

4.1. Expression< T > osztálysablon-referencia

```
#include <Expression.hpp>
```

4.1.1. Publikus tagfüggvények

1. **Expression** (std::string infix)
2. **Expression** (T value=T())
3. T **eval** () const
4. std::string **getInfix** () const
5. std::string **getPostfix** () const
6. std::string **getPrefix** () const
7. **Expression operator+** (const **Expression** &e)
8. **Expression** & **operator+=** (const **Expression** &e)
9. **Expression operator-** (const **Expression** &e)
10. **Expression** & **operator-=** (const **Expression** &e)
11. **Expression operator*** (const **Expression** &e)
12. **Expression** & **operator*=** (const **Expression** &e)
13. **Expression operator/** (const **Expression** &e)
14. **Expression** & **operator/=** (const **Expression** &e)

4.1.2. Részletes leírás

4.1.2.1 template<typename T>

4.1.2.2 class Expression< T >

Kifejezések osztálya

4.1.2.2.1 Paraméterek:

T	- adattípus

4.1.3. Konstruktorok és destruktorok dokumentációja

4.1.3.1 template<typename T> Expression< T >::Expression (std::string *infix*)

Infix stringbõl készít kifejezést.

Sajnos későn jutott eszembe, hogy a negatív számokat is kezelnie kell a programnak, így gyors megoldásnak azt választottam, hogy a megadott stringben space-el kötelező elválasztani a számokat egymástól és az operandusoktól. Így feldolgozáskor el tudjuk dönteni, hogy a „-” jel operátor vagy az operandushoz tartozó előjel. A tizedeshelyet ponttal lehet jelölni.

Pl: 3 + 5 * (7.6 + 3.9) * -8

4.1.3.1.1 Paraméterek:

infix	- Infix formátumban megadott string, az operandusokat és az

	operátorokat egymástól space választja el.

template<typename T> Expression< T >::Expression (T *value* = T( ))Értékbõl készít kifejezést

4.1.3.1.2 Paraméterek:

value	- ezt az értéket rendli a kifejezéshez

4.1.4. Tagfüggvények dokumentációja

4.1.4.1 template<typename T> T Expression< T >::eval () const

Kiértékeli a kifejezést

4.1.4.1.1 Visszatérési érték:

- A kifejezés értéke

4.1.4.2 template<typename T> std::string Expression< T >::getInfix () const

infix formátumba konvertálás

4.1.4.2.1 Visszatérési érték:

- String, infix formában

4.1.4.3 template<typename T> std::string Expression< T >::getPostfix () const

postfix formátumba konvertálás

4.1.4.3.1 Visszatérési érték:

- String, postfix formában

4.1.4.4 template<typename T> std::string Expression< T >::getPrefix () const

prefix formátumba konvertálás

4.1.4.4.1 Visszatérési érték:

- String, prefix formában

4.1.4.5 template<typename T> Expression Expression< T >::operator* (const Expression< T > & *e*)

Kétkifejezést szoroz

4.1.4.5.1 Visszatérési érték:

- kifejezések szorzata

4.1.4.6 template<typename T> Expression& Expression< T >::operator*= (const Expression< T > & *e*)

Kétkifejezést szoroz helyben

4.1.4.6.1 Visszatérési érték:

- kifejezések szorzata helyben

4.1.4.7 template<typename T> Expression Expression< T >::operator+ (const Expression< T > & *e*)

Kétkifejezést összead

4.1.4.7.1 Visszatérési érték:

- kifejezések összege

4.1.4.8 template<typename T> Expression& Expression< T >::operator+= (const Expression< T > & *e*)

Kétkifejezést összead helyben

4.1.4.8.1 Visszatérési érték:

- kifejezések összege helyben

4.1.4.9 template<typename T> Expression Expression< T >::operator- (const Expression< T > & *e*)

Kétkifejezést kivon

4.1.4.9.1 Visszatérési érték:

- kifejezések kulonbsege

4.1.4.10 template<typename T> Expression& Expression< T >::operator-= (const Expression< T > & *e*)

Kétkifejezést kivon helyben

4.1.4.10.1 Visszatérési érték:

- kifejezések kulonbsege helyben

4.1.4.11 template<typename T> Expression Expression< T >::operator/ (const Expression< T > & *e*)

Kétkifejezést eloszt egymassal

4.1.4.11.1 Visszatérési érték:

- kifejezések hanyadosa

4.1.4.12 template<typename T> Expression& Expression< T >::operator/= (const Expression< T > & *e*)

Kétkifejezést eloszt egymassal helyben

4.1.4.12.1 Visszatérési érték:

- kifejezések hanyadosa helyben

Az általam a konverziókhoz és a kiértékeléshez használt algoritmusok többször elvárják, hogy a postfix kifejezésen balról jobbra (avagy a stackban lentről fölfelé), haladjak ez elemeket. Mivel az Expression osztályban a kifejezés tárolását stackben oldottam meg, azt a nem túl elegáns megoldást alkalmaztam, hogy létrehozom a stack egy tükörképét, és ezen a másolaton már tudok dolgozni. Így utólag lehet, hogy érdemesebb lett volna vektort használni, vagy készíteni a stacknek egy olyan iterátor interfészt, amit csak az Expression osztály tud használni.

4.2. Tesztprogram bemutatása

A main.cpp-ben levő tesztprogram a stdin-ről beolvasott számok alapján különböző teszteket hajt végre. A teszteléshez használok adatfájlokat is (values.txt, expcode.txt). A tesztelés közben esetlegesen dobott kivételeket a tesztprogram elkapja.

4.2.1. Az egyes teszteseteket megvalósító függvények:

Void stacktest()

A stack kezelést teszteljük, az outputra írt infók alapján szemrevételezéssel ellenőrizhetjük a helyes működést (pop, push, megfordítás).

Void expressiontest()

Kifejezések létrehozása értékből, infix stringből, postfix-, infix- és prefix stringgé alakítás. Output kiírja az elvárt és a kapott kimenetet.

Void eval()

A kézzel stringként és értékként megadott kifejezések kiértékelésének tesztelése, kiírja az elvárt és a kapott értéket.

Void stringconvert()

Az Expression osztály által használt stringátalakító függvények helyességét ellenőrizhetjük(postfix → infix, postfix → prefix) szemrevételezéssel

void negativ()

Helyesen kezeli-e a negítiv számokat az infix stringben?

Void muveletek()

Két expression összefűzése adott művelettel. A kapott kiértékelés alapján láthatjuk, hogy helyes a működés

void grandeval()

Az általam véletlenül generált values.txt és expcodes.txt fájlokból beolvas száz db kifejezés stringet, és ezek kiértékelését összehasonlítja a 100 db elvárt értékkel. Ha

találna a program nem egyezést, jelezné.

Void grandinfix()

A véletlenült generált 100 infix stringből létrehoz 100 kifejezést, és kiszámítja az ezeknek megfelelő infix stringeket. Az így kapott stringek formátuma nem teljesen egyezik meg a bemeneten levőkkel, így ezeket még egyszer átengedjük a konstruktoron. Ha most kérdezzük le az infix stringeket, ezeknek már meg kell egyezniük az előző, közbülső stringekkel. Ha minden megegyezik, az átalakítás helyes.

5. Tesztelés

5.1. Interfész és funkcionális teszt

A fentiekben bemutatott tesztek tartalmaznak példákat double-ökkel, intekkel, tizedespontokkal, zárójelekkel dolgozó stringekből Expressionok létrehozására, kiértékelésére, stringgé alakítására. A tesztekből meggyőződhetünk a helyes funkcionális működésről, és mivel a kód lefut, az interfész helyességéről is.

5.2. Memóriakezelés tesztje

A main.cpp-be includeoltam a memtrace.h fájlt. Mivel az újabb gcc nem tudja kezelni a memtrace-t a jportára hagyatkoztam, ami nem jelzett memóriaszivárgást.

6. Felhasználói dokumentáció

Főbb funkciók bemutatása

6.1. Expression létrehozása

6.1.1. értékből:

Expression<int> e(4);

Expression<double> e2(5.6);

A template paraméternek csak a kiértékelésnél van jelentősége.

6.1.2. Stringől

Expression<int> e3(„3 + 4 * (5.5 / -4)”);

Minden karakter között space kell hogy legyen, kivéve a tizedes pontnál és a negatív előjelnél.

6.2. Kiértékelés:

e3.eval();