



UNIVERSITEIT VAN AMSTERDAM

PROGRAMMEERTALEN

WEEK 7 - C++

BAS TERWIJN

Extended Evaluator

Deadline: see Canvas

Inleiding

Het bestand “evaluator.zip” bevat de source code van een evaluator waarmee postfix expressies kunnen worden uitgevoerd die van de standard input stream worden ingelezen. Compileer deze code met **make** (negeer voor nu de warnings) en test de evaluator met:

```
$ echo "./data/d1.txt ./data/d2.txt + ./data/d1.txt *" | ./evaluator
13.75
```

Deze output is het resultaat van $(d1 + d2) * d1$ waarbij respectievelijk de waarde $d1$ en $d2$ uit bestand “./data/d1.txt” en “./data/d2.txt” wordt gelezen. In de onderstaande opdrachten breiden we deze evaluator uit.

Opdracht: Individueel

Een expressie in postfix vorm, waarbij de operator na de argumenten komt ($ab+$), is makkelijker te verwerken dan een expressie in infix vorm, waarbij de operator tussen de argumenten in staat ($a + b$). Maar voor een mens is de infix vorm makkelijker te lezen. Voltooi daarom “individual.h” zodat het een infix expressie van de standaard input stream kan lezen, de infix expressie omzetten naar postfix expressie, en het resultaat naar standaard output schrijft. Zorg daarbij dat de argumenten en operators door whitespaces gescheiden worden zoals bijvoorbeeld:

```
$ echo "(a+b)+c*d" | ./individual
a b + c d * +
```

Zodat op deze manier infix expressies naar postfix expressies kunnen worden vertaald en verwerkt door de evaluator:

```
$ echo "(./data/d1.txt + ./data/d2.txt) * ./data/d1.txt" | ./individual | ./evaluator
13.75
```

Een expressies kan bestaan uit de volgende speciale tekens en bestandsnamen (in bestandsnamen mogen geen speciale tekens of whitespaces voorkomen):

- ‘~’: het prefix min-teken, bijvoorbeeld “~5” wat we interpreteren als “min 5” of “-5”
- ‘”’: prefix operator voor matrix transpose, bijvoorbeeld “'5” wat voor scalar 5 gelijk is aan 5.
- ‘*’: infix operator voor multiplicatie, bijvoorbeeld “5*4”
- ‘+’: infix operator voor optellen, bijvoorbeeld “5+4”
- ‘-’: infix operator voor aftrekken, bijvoorbeeld “5-4”
- ‘(en ’)’: voor beïnvloeden van de bewerkings-volgorde, bijvoorbeeld “(5+4)*3”

Deze opdracht lijkt sterk op een opdracht in vak Datastructuren, maar hier schrijven we geen eigen stack maar gebruiken we “std::stack” uit de C++ standard template library welke een verzameling nuttige datastructuren bevat. Een beschrijving van een algoritme voor omzetten van infix naar postfix vind u hier:

- Youtube video: <https://www.youtube.com/watch?v=LQ-iW8jm6Mk>
- Wikipedia: https://en.wikipedia.org/wiki/Shunting-yard_algorithm

Opdracht: Matrix (Ontwikkeland)

In bestand “evaluator.cpp” heeft variabele “evaluator” het type “Evaluator<double,0>”, een template class met twee template argumenten. Het tweede template argument “Log_level” is een non-type template argument van type “int”.

Als we hier bijvoorbeeld i.p.v. waarde 0 de waarde 2 kiezen zal de evaluator tijdens het uitvoeren van expressies een log printen van stappen die worden uitgevoerd, dit kan handig zijn bij debuggen. Het eerste template argument “Argument_type” bepaalt hoe de inhoud van een bestand geïnterpreteerd wordt. Omdat er “double” staat wordt de inhoud van bestanden ingelezen als double. Verander het in “int” om de inhoud te lezen als int en zie het verschil in resultaat bij uitvoeren van de bovenstaande expressie.

Voltooi nu “Matrix.h” zodat we voor “Argument_type” type “Matrix” kunnen kiezen waardoor de evaluator ook bestanden met matrices van willekeurige grootte kan verwerken zoals bijvoorbeeld “./data/m1.txt”. In een bestand wordt een matrix gerepresenteerd door regels met comma gescheiden elementen. Bijvoorbeeld:

```
$ echo " ' ./data/m1.txt * ./data/m1.txt " | ./individual | ./evaluator
20.57,26.62,32.67
26.62,35.09,43.56
32.67,43.56,54.45
```

Tip: Test uw Matrix class eerst met “Matrix_operators_test.cpp” voordat u deze samen met de evaluator test.

Opdracht: MatrixT (Vaardig)

De Matrix class representeert een matrix van type double. Class MatrixT in “MatrixT.h” is een matrix waarbij we met een template argument het type kunnen kiezen. Voltooi “MatrixT.h” zodat we met bijvoorbeeld “MatrixT<int>” als “Argument_type” ook een bestand als matrix van integers kunnen lezen:

```
$ echo " ' ./data/m1.txt * ./data/m1.txt " | ./individual | ./evaluator
17,22,27
22,29,36
27,36,45
```

Tip: Test uw MatrixT class eerst met “MatrixT_operators_test.cpp” voordat u deze samen met de evaluator test.

Vermijd code duplicatie tussen “Matrix.h” en “MatrixT.h” door class Matrix te herschrijven en te definiëren in termen van class MatrixT<T>.

Opdracht: Str (Competent)

Voltooi “Str.h” zodat we class “Str” als “Argument_type” kunnen kiezen. De Str class moet de waarden als string lezen en het resultaat van een expressie is een string met de uitgeschreven expressie zoals bijvoorbeeld:

```
$ echo "(./data/d1.txt + ./data/d2.txt) * ./data/d1.txt" | ./individual | ./evaluator
(2.5+3)*(2.5)
```

Hierbij is het overkillig gebruik van haakjes ‘(’ en ‘)’ geen probleem zolang de expressie logisch maar correct is omdat deze toch wegvallen als we later een infix naar postfix omzetting zouden doen.

Opdracht: MatrixT<Str> (Gevorderde)

Zorg dat we ook class “MatrixT<Str>” als “Argument_type” kunnen kiezen welke ook matrix expressies uitschrijft zoals bijvoorbeeld:

```
$ echo " ' ./data/v1.txt * ~./data/v1.txt " | ./individual | ./evaluator
(a)*(-(a))+(e)*(-(e)),(a)*(-(b))+(e)*(-(f)),(a)*(-(c))+(e)*(-(g)),(a)*(-(d))+(e)*(-(h))
(b)*(-(a))+(f)*(-(e)),(b)*(-(b))+(f)*(-(f)),(b)*(-(c))+(f)*(-(g)),(b)*(-(d))+(f)*(-(h))
(c)*(-(a))+(g)*(-(e)),(c)*(-(b))+(g)*(-(f)),(c)*(-(c))+(g)*(-(g)),(c)*(-(d))+(g)*(-(h))
(d)*(-(a))+(h)*(-(e)),(d)*(-(b))+(h)*(-(f)),(d)*(-(c))+(h)*(-(g)),(d)*(-(d))+(h)*(-(h))
```

Opdracht: Algorithms (Expert)

Voor de implementatie van Matrix en MatrixT zou u *raw loops* kunnen gebruiken, maar dat resulteert in code met een laag abstractie-niveau. Dit lage abstractie-niveau resulteert vaak in een hogere kans op bugs en een lagere leesbaarheid. Gebruik voor de Matrix implementatie nu zoveel mogelijk, in plaats van *raw loops*, functies uit C++ algorithms en aanverwante libraries:

<https://en.cppreference.com/w/cpp/algorithm>

Misschien dat u deze CppCon-2018 video hierbij nuttig vindt:

Youtube video: <https://www.youtube.com/watch?v=2olsGf6JIkU>

Als u echt ver wilt gaan zou u hierbij ook bijvoorbeeld een `Row_const_iterator` class kunnen definiëren om over de rows van een matrix te kunnen itereren.

Vind u dat de code-kwaliteit is toegenomen door gebruik van algorithms in plaats van *raw loops* bij de implementatie van deze matrix class? Voeg eventueel uw antwoord als commentaar bovenin toe.

Foutafhandeling

In het algemeen is foutafhandeling natuurlijk belangrijk, maar omdat we hier alle aandacht op de C++ concepten willen vestigen mag u aannemen dat alleen geldige expressie worden ingevoerd en dat ook alle bestanden een geldig formaat hebben. U hoeft dus geen code te schrijven die dit controleert. Controleer wel of de dimensies van matrices kloppen bij het uitvoeren van een expressie en gebruik voor het melden van deze en mogelijk andere problemen de “`Evaluator_exception`” class in “`evaluator_exception.h`”.

Template foutmeldingen

Een compiler kan erg lange en complexe foutmeldingen geven met betrekking tot template arguments. Vaak staat de meest nuttige informatie aan het begin en/of het einde van zo’n foutmelding. Deze foutmeldingen komen vaak voort uit een functie of operator die wordt aangeroepen op de template parameter terwijl deze functie of operator niet (juist) gedefinieerd is voor een gegeven template argument. Let hierbij vooral ook op het ‘const’-zijn van variabelen en functies. In “`Matrix.h`” en “`MatrixT.h`” zijn als voorbeeld de benodigde functies en operators al gegeven. Voor class `Str` in “`Str.h`” zult u dit zelf moeten oplossen.

Inleveren

Voor de individuele deadline lever alleen bestand `individual.h` in, voor de team deadline lever de `Matrix.h`, `MatrixT.h` en `Str.h` bestanden in.