Linköpings Universitet
Institutionen för datavetenskap (IDA)
UPP-gruppen 2022-08-26

Kalkylator

Bakgrund

Förr i tiden, när att räkna i huvudet eller för hand inte räckte till, använde man räknestickor och abakusar. Komplexa beräkningar kunde ta åratal av manuella beräkningar i anspråk[1]. I mer modern historia har tunga beräkningar utförts av arbetsgrupper med yrkesrollen "Computer"[2]. Nästa alla "Computers" var kvinnor. När elektroniska beräkningsmaskiner så småningom infördes som hjälpmedel var det dessa "computers" som blev de första programmerarna[3]

Idag använder vi algoritmer implementerade som datorprogram för att utföra dessa beräkningar. Ska vi beräkna något till vardags förlitar vi oss på digitala miniräknare. Vad man kanske inte tänker på är att även miniräknaren innehåller ett program för att utföra beräkningarna. Funderar du ett slag på vad din TI'82 kan göra inser du att programmet den använder är väldigt avancerat.

I denna labb ska vi skapa en programmodul (bestående av flera klasser) för att enkelt kunna hantera matematiska uttryck. Sedan ska vi använda modulen till att implementera en enklare kalkylator. Vårt mål är att göra det så enkelt som möjligt att bygga ut uttryck med flera beräkningssätt, och göra det enkelt att bygga och bygga ut vår kalkylator. Vill vi återanvända vår uttrycksmodul i flera program ska det inte vara svårare än att inkludera en headerfil och kompilera in motsvarande implementationsfil. Att integrera beräkning av uttryck i andra program är inte ovanligt, den välkända betalappen Swish är ett exempel. Googles sökfält är ett annat exempel.

Vi kommer att lösa labben en del i taget. Varje del kommer bygga vidare på den tidigare. Det gör att ni kan behöva ändra kod ni tidigare skrivit. Ha hela tiden i åtanke att skriva koden så att det blir så enkelt som möjligt att göra om samma typ av tillägg eller förändring. Finns det ett sätt att skriva koden så en framtida ändring eller bugfix bara behöver utföras på en plats i stället för fem ställen ska ni organisera koden så det blir möjligt.

Det går så klart bra att läsa framåt i instruktionen för att se vilka utmaningar som kommer och planera för dem. Det är dock inget krav. Att det kan uppstå nya krav reflekterar hur projektarbeten brukar se ut. Vi vet inte vad för utmaningar som kommer härnäst utan kan bara arbeta för göra framtida utmaningar enklare att hantera.

[1] `https://en.wikipedia.org/wiki/Chronology_of_computation_of_pi`

[2] `https://en.wikipedia.org/wiki/Computer_(job_description)`

[3] `https://en.wikipedia.org/wiki/Kathleen_Antonelli`, `https://en.wikipedia.org/wiki/Betty_Holberton`, `https://en.wikipedia.org/wiki/Marlyn_Meltzer`, `https://en.wikipedia.org/wiki/Ruth_Teitelbaum`, `https://en.wikipedia.org/wiki/Jean_Bartik`, `https://en.wikipedia.org/wiki/Frances_Spence` och inte minst `https://en.wikipedia.org/wiki/Katherine_Johnson`.

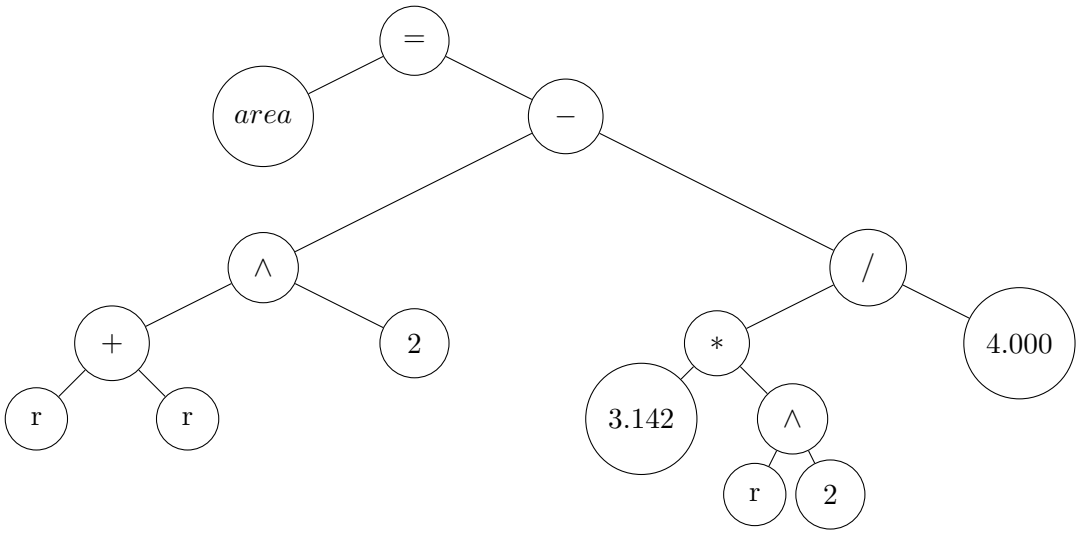
Steg 1: Objektorienterad analys och design (OOA och OOD)

I uppgifterna i steg 1 *ska* ingen kod skrivas. Du ska använda dig av:

- Objektorienterad analys
- Objektorienterad design
- CRC-kort eller motsvarande information
- UML Klassdiagram

Inledning: Representation av uttryck

Det finns flera sätt att beskriva matematiska uttryck. Det vi är vana vid kallas *infix* och är den variant som är krångligast att hantera. Två alternativ som går att beräkna med avsevärt enklare regler är *prefix* och *postfix*. Vi ska dock hämta inspiration från hur kompilatorn representerar programkod, där ju beräkning av uttryck är en del. I denna representation beskrivs uttrycket som ett träd bestående av noder och kopplingar mellan noder. Denna representation lämpar sig mycket väl för att enkelt transformera uttryck till prefix, infix och postfix, men även för att förenkla(delberäkna) uttryck så mycket som möjligt. Här kommer ett exempel på hur infixet $area = (r + r)^2 - pi * r^2/4$ ser ut i trädform. (Kuriosa: Beräkningen ger ytan som blir kvar om vi tar bort en Pacman med radien r ur en kvadrat.)



Figur 1: Ett uttryckträd

För att kunna prata om trädet är det bra att först klargöra lite terminologi. Trädet består av *noder*. Strecken mellan noderna visar dess relation till andra noder. Ett streck från en ovanliggande nod till en underliggande visar att den ovanliggande noden är förälder till den undre. Omvänt blir då den undre barn till föräldern. Högst upp i figuren är *roten* (=). Roten är den nod som inte har någon förälder

över sig. De noder längst ut på grenarna som inte har några barn under sig kallas *löv*. I figuren kan vi identifiera flera olika typer av noder. Några är *operander* och andra är *operatorer*. Både operander och operatorer finns i flera olika varianter.

Värt att observera är att trädet är rekursivt uppbyggt. Tittar vi på flyttalsnoden (4.000) så är den ensam ett eget uttryck. Dess förälder (/) är ett uttryck som består av ett (*)-uttryck till vänster och ett (4.000)-uttryck till höger. Och tittar vi på roten (=) så representerar den det fullständiga uttrycket.

Vårt mål till slut är att kunna behandla hela uttryck lika enkelt som vi kan behandla heltal eller strängar. Det vi ska kunna göra med ett skapat uttryck är att konvertera det till prefix-, infix- eller postfixsträng, kopiera det, beräkna hela uttrycket och förenkla(delberäkna) det. Du hittar exempel på hur de olika operationerna ska gå till enligt rekursivt tankesätt i Exempel-avsnittet.

För att komma fram till en bra objektorienterad representation är ditt tankesätt mycket viktigt. Tänk på varje objekt som en självständig enhet. Ett objekt kan tillhandahålla tjänster utifrån vad det vet om sig själv. Ett objekt ska inte veta något om andra objekt, utöver vad vi kan be ett annat objekt att utföra. Alla operationer som ska kunna utföras ska tillhöra ett för operationen relevant objekt.

Exempel: Tänk dig att vi ska bygga en lekplats. Vi identifierar snabbt objekten rutschkana, gunga, sandlåda och klätterställning. Nu tänker du kanske att vi beställer hem virke och en snickare som vet hur man sätter ihop allt? **Nej!** Vi beställer hem ett objekt "gunga" och säger "skapa dig med färgen gul och typen gummidäck". Så beställer vi hem en rutschkana och säger "skapa dig röd med fallhöjd 2 meter". Till sandlådan kanske vi säger "skapa dig kvadratisk med kornstorlek 1mm". Poängen är att varje objekt själv vet hur det ska skapas utifrån en inbyggd ritning (klassen). Samma idé gäller när objektet används, alla vet *vad* objekten kan användas till (swing.ride(), slide.ride()) men det är bara gungan som vet att den orsakar en pendelrörelse, och det är bara rutschkanan som vet hur snabb den är och hur mycket vatten som väntar längst ned.

Uppgift 1 Identifiera objekt och operationer

1. Studera figuren och läs igenom texten för representationen ovan.
2. Identifiera alla olika typer av noder i figuren.
3. Namnge varje nodtyp du identifierat.
4. Fundera igenom *vad* varje nod behöver be sina barnnoder om. Tänk i termer av resultat som behövs. Undvik fundera på hur resultatet ska produceras. Fundera igenom vad varje nod behöver för att utföra sin grej utan att veta något om hur andra noder fungerar.
5. Fundera igenom *vilka* operationer varje nod behöver tillhandahålla. Detta ges av ovan. Att tänka omvänt gör dock att vi ibland kommer på andra saker. Namnge varje operation du kommer fram till så enhetligt och generaliserat som möjligt. Tänk på varje nod både i termer av sig själv och i termer av ett generellt uttryck.
6. Fundera på vilka attribut(egenskaper) varje nodtyp behöver hålla reda på för att kunna lösa sin grej.

7. Behöver någon nod känna till andra noder? Vad behöver den i så fall veta om dem? I objektorientering strävar varje objekt efter att veta så lite som möjligt om andra objekt. Objekt strävar även att hålla all information om det egna objektet för sig själv. Båda strävansmålen ger oss mycket bättre möjlighet ändra på objekt senare utan att andra objekt påverkas.

Varje nodtyp du kommit fram till är nu kandidat till att bli en egen klass. Skriv ned det du kommer fram till som ett CRC-kort för varje klass. CRC står för Class, Responsibility, Collaborators. Det är ett sätt att sammanfatta en klass med klassens namn (ett ord), vad den har för uppgift (en mening, ev attribut) och namn på ev klasser den behöver samarbeta med. Du kan istället skriva ned motsvarande data i tabellform för att lättare kunna jämföra de olika klasserna.

Kontroll

Nu ska du ha kommit fram till klasser för 3 lövnodtyper och mellan 6 och 9 andra nodtyper. Om du har fler, fundera på om några av de du har egentligen beskriver samma sak. Om du har färre, fundera på om du missat något i figuren som skiljer sig åt mellan olika noder.

Uppgift 2 Strukturera och rita klassdiagram

- Gå igenom dina klasser och jämför dem. Finns det några klasser som har vissa egenskaper gemensamt med andra klasser? Vi vill inte behöva upprepa gemensamma delar i varje klass, så inför basklasser för det som är gemensamt.
- Gå igenom dina klasser och fundera på om det finns någon operation som är gemensam för flera klasser. Tänk bort *hur* operationen ska utföras och fokusera på *vad* operationen ska utföra. T.ex. betraktar vi operationen “skriv ut” som samma operation även om två klasser ska utföra den på olika sätt. Placera gemensamma operationer i gemensam basklass.
- Fundera på om det finns klasser som tillhör en gemensam kategori. Sammanför även dem i en gemensam basklass även om du (än så länge) inte ser några gemensamma egenskaper eller operationer.
- Se över egenskaperna i varje klass. Behöver du verkligen lagra dem? Om en egenskap är direkt given utifrån klassens namn behöver vi inte lagra den i klassen.
- Rita upp dina klasser i ett klassdiagram.

Kontroll

Nu bör du ha en arvshierarki med klasser i tre nivåer. En basklass, två härledda “mellan”-klasser, och 9 lövklasser. Egenskaperna i varje klass ska skilja sig åt i antingen syfte eller typ, så inga egenskaper är upprepade. Inga egenskaper som är direkt givna av klassens namn ska finnas. Tänker vi oss t.ex. en klass “Modulus” behöver vi inte någon datamedlem för att veta att modulus utförs med operatorn `%`. Alla operationer som behöver utföras ska vara del av en eller flera klasser. Inga operationer ska behöva känna till hela uttryckskrädet.

Bonus: Användning av git

Sätt upp ett repository för labben på `gitlab.liu.se` och checka in ditt klassdiagram. Checka sedan in lösningen på varje uppgift vartefter du jobbar vidare. Turas om med din kompanjon att vara både den som skriver koden och den som checkar in koden. Båda ska ha ungefär lika många commits när ni är klara. Använd git på kommandoraden. Du behöver följande kommandon för de vanliga uppgifterna i git:

- `git status`
- `git add`
- `git commit`
- `git push`
- `git pull`

När du behöver ångra eller göra något utöver det vanliga finns dokumentation på `https://git-scm.com/book/en/v2`. När git vill låta dig redigerar en fil öppnas ibland ett textverktyg. Sätt miljövariabeln EDITOR för att välja ditt favoritverktyg, t.ex. `export EDITOR=/usr/bin/emacs`.

Steg 2: Implementation och test av klasshierarki

I uppgifterna i steg 2 *ska* du utöver vad du lärt dig i tidigare laborationer använda dig av:

- Klasser med relationen arv(specialisering)
- Polymorfa medlemsfunktioner
- Abstrakta klasser och rent virtuella (pure virtual) funktioner
- Rekursivt tankesätt för implementation av medlemsfunktioner
- Objektorienterat tankesätt för självständiga klasser med tillhörande operationer
- `new` och pekare för att allokera minne till objekt
- `std::logic_error` för generering av undantag
- Förbjuda användning av kopieringskonstruktor och tilldelningsoperator

I uppgifterna i steg 2 får du *inte* använda dig av:

- Villkorssatser (if, else, switch). Använd istället polymorfi.
- Upprepningssatser (for, do, while). Använd istället rekursivt tänk.
- Fria funktioner. Använd istället medlemsfunktioner i relevant klass.

Inledning: Objektorienterad programmering

Inom objektorienterad programmering strävar vi efter att dela in programmet i små självständiga moduler. Varje modul ska ha ett tydlig ansvarsområde (high cohesion), fylla sin uppgift så enkelt som möjligt och vara relativt isolerad (low coupling). Ju närmare vi kommer desto lättare att utveckla modulerna självständigt utan att minsta ändring får stora konsekvenser för många andra moduler.

Det är lätt att fastna i tankesättet “vad behöver jag(eller modulen jag skriver) för att lösa problemet?”. Du riskerar då få en stor “gud”-modul som löser “allt” och inte når de objektorienterade målen. Tänk istället “vilka andra moduler kan jag(modulen jag skriver) be lösa de delar av uppgiften som hör till dem mer än till mig?”.

Exempel: I uppgifterna ska noder *inte* behöva veta något om hur andra noder beter sig för varken utskrift eller beräkning. Be istället noderna göra sin grej på sitt sätt. Det är då viktigt att noderna skapas med all information de behöver ha tillgång till senare. Det är också viktigt att operationer som åstadkommer samma sak på olika sätt (beräknat värde, postfix-sträng) kan anropas på ett enhetligt sätt oavsett nodtyp (polymorfi).

Uppgift 3

Vi ska börja med klasserna för att representera de noder som behövs för att kunna använda heltal, flyttal, addition och subtraktion. Vi använder testdriven utveckling (TDD). Fundera igenom hur respektive objekt ska konverteras till postfix och hur det ska testas. Använd minnesallokering med `new` och pekare för att hålla reda på noder, både i testprogrammet och i de noder som behöver hålla reda på andra noder.

Skriv tester för att skapa de olika noderna enskilt och be varje nod konvertera sig till postfix-sträng. Använda catch för att jämföra resultatet med det du förväntar få. Det finns ett givet testprogram att utgå från. Gör ett test för en klass och dess postfix-konvertering först, och gå vidare till nästa klass och postfix när det fungerar. Lägg sedan till test och implementation av beräkning av varje enskild klass, en i taget. Du hittar exempel på hur konvertering och beräkning går till i Exempel-avsnittet. Målsättning:

- Implementera testfall för nedan klasser och funktioner.
- Implementera de klasser som behövs för att kunna använda heltal, flyttal, addition och subtraktion.
- Implementera en funktion för att konvertera varje nod till en postfix-sträng. Resultatet ska vara en `std::string`. Ett heltal ska aldrig ha några decimaler. Ett flyttal ska alltid ha exakt 3 decimaler. Addition görs om till sträng genom att först göra om vänsternoden till sträng, sedan högernoden, och sedan mata ut ett `+`-tecken. Separera varje del med exakt ett blanksteg.
- Implementera en funktion för att beräkna varje nod. Resultatet ska vara en `double`. Ett heltal beräknas genom att helt enkelt returnera heltalsvärdet. Addition beräknas genom att först beräkna vardera undernod och sedan returnera summan av dem.
- Kontrollera att testfallen lyckas och är kompletta.

Uppgift 4

När kompilatorn varnar om “class X have pointer data members but does not override ...” kan du förbjuda att kopieringskonstruktorn och tilldelningsoperatorn kompilatorn pratar om används. Det gör att funktionerna inte kan råka användas och varningen försvinner. Vi kommer titta närmare på minneshanteringen i senare uppgifter. Just nu räcker det du lägger till följande deklarationer i klassen X:

```
X(X const&) = delete;
X& operator=(X const&) = delete;
```

Kontroll

Har du bara en stor funktion för strängkonvertering och en för beräkning är du på fel spår (och har dessutom använt en lång rad villkor för att testa nodtyp). Den lösningen gör att varje funktion måste känna till allt om alla klasser. Det bryter mot tanken att kunskap om klassen ska vara isolerad inom klassens egna funktioner.

Vi förväntar att du nu har implementerat fyra funktioner för beräkning och fyra funktioner för konvertering till postfix-sträng. När du jämför funktionerna för konvertering till postfix ser du att två av dem är nästan helt identiska. Så kan vi inte ha det. Vi löser koduppprepningen i nästa uppgift.

Uppgift 5 Refaktorering

Inför en ny funktion som ger tillbaka operatorsymbolen för respektive operator, och använd den funktionen för att göra de två snarlika postfix-funktionerna helt identiska. Nu kan du flytta operatorernas implementation av konvertering till postfix till gemensam basklass. När du sedan lägger till nästa operatorklass får du konvertering till postfix gratis!

Uppgift 6

Skriv testfall och implementera klasser för multiplikation, division och exponentiering. Kontrollera att beräkning och konvertering till sträng fungerar för samtliga klasser. Tänk på att ^ i C++ utför bitvis XOR. För exponentiering behöver du funktionen `pow` från `<cmath>`. Även `fmod` är användbar för att kontrollera om ett flyttal har några decimaler.

I de fall en beräkning inte kan utföras med ett reellt tal som resultat ska beräkningen kasta ett `std::logic_error`. Detta gäller speciellt division med noll, negativa tal med flyttal som exponent, och noll med negativ exponent. *För att kontrollera om beräkningen kan utföras har du tillstånd använda villkorssatser.*

Uppgift 7 Infix

- Använd TDD och lägg till ett test i taget för nedan funktioner.
- Visst vore det trevligt att kunna se uttrycken som infix, så som vi är vana? Lägg till funktioner för att konvertera till en infix-sträng. Tänk på vad du lärt dig så du inte gör mer än en implementation för operander (en operand ha samma prefix, infix och postfix) och en implementation för operatorer. Tänk på att infix behöver parenteser för att beräkningsordningen ska bli rätt. Det är okej att lägga till parenteser runt varje beräkning.
- Lägg till konvertering till prefix. I prefix skrivs först operatorsymbolen, sedan vänsternoden och sist högernoden.
- Kontrollera att alla gamla och nya tester lyckas.

Uppgift 8 Sammansatta uttryck

Om du inte redan gjort det, använd dina nodklasser för att bygga upp sammansatta uttryck i ditt testprogram. Refaktorera och kontrollera. Gå igenom alla testfall du har och fundera på om det finns någon kodrad som aldrig kommer exekveras när du kör dem. Lägg till tester så all din kod testas.

Bonus: Användning av git

Har du glömt checka in uppgifterna i git? Ingen fara. Med git kan du välja vilka delar av en fil du ska lägga till i varje commit. Det är bra när du jobbat på flera features eller buggar samtidigt, men vill checka in koden för varje feature eller bug separat. Se "Staging Patches" med kommando `git add --patch` på `https://git-scm.com/book/en/v2/Git-Tools-Interactive-Staging`. Detta låter dig checka in varje uppgift för sig även om allt redan är klart. Om git inte låter dig dela upp en ändring med "s", läs vidare på `https://drewdeponte.com/blog/git-add-patch-wont-split/`.

Frivilligt

Känner du att detta med att lägga till nya operatorer är enkelt? Bra! då har du lyckats med din kod. Bevisa det genom att lägga till en operator för att beräkna modulus (%). Modulus ger resten vid en division där kvoten är ett heltal, tar vi $7.5\%3.1$ får vi kvoten 2 och resten 1.3 ($2 * 3.1 + 1.3 = 7.5$). Vill du gå helt bananas? Prova lägga till en villkorsoperator (?) där högernoden endast beräknas om vänsternoden är nollskild.

Steg 3: Uttrycksklass

I uppgifterna i steg 3 *ska* ska du utöver det du lärt dig i tidigare laborationer och uppgifter använda dig av:

- std::stack för konvertering från postfix till uttrycksträd
- `try-catch`-satsen för att fånga undantag i ditt huvudprogram

Inledning: Top down design av gränssnitt

Nu har vi skapat klasser för att representera ett uttrycksträd med de vanliga räknesätten och taltyperna. Men om vi tittar på klasserna ser vi att de inte är speciellt lätt att använda dem. Det ska vi åtgärda nu genom att skapa en klass med ett gränssnitt som är enkelt att använda och döljer alla komplicerade noder. Vi ska senare även låta den klassen ta hand om all minneshantering som vi ignorerat så här långt. Vi börjar med att föreställa oss hur vi vill kunna använda uttryck i ett huvudprogram:

```
#include <iostream>
#include "expression.h"

using namespace std;

int main()
{
    string line;

    while ( getline(cin, line) )
    {
        Expression e{line};

        cout << e.to_string()
             << e.evaluate() << endl;
    }
    return 0;
}
```

Svårare än så vill vi inte att det ska vara! Vi skapar ett uttryck `Expression` utifrån en given inmatning och sedan kan vi enkelt både skriva ut och beräkna uttrycket. För att använda uttryck ska vi inte behöva känna till något om alla de klasser vi redan byggt. Uttrycksklassen tar hand om att bygga upp vårt uttrycksträd med hjälp av de klasser som redan finns och håller reda på roten till uttrycksträdet.

Uppgift 9 Gränssnitt för uttryck

Designa gränssnittet till uttrycksklassen. Tänk igenom vilka konstruktorer, datamedlemmar och medlemsfunktioner som behöver finnas. Det ska inte vara möjligt att skapa uttryck som inte fungerar. När vi väl skapat ett uttryck via konstruktorn ska det alltid fungera att skriva ut och beräkna (det är okej att beräkningen genererar undantag ni kastar). Tänk speciellt igenom hur tomma uttryck ska

hanteras. Du kan förbjuda att tomma uttryck skapas, du kan ersätt tomma uttryck med "0", eller du kan låta klassen hantera att uttryckskträdet ibland är tomt. Fundera på vad som är smidigast. Tänk på att om du senare vill kunna överlagra inmatningsoperatorn `operator>>` för uttryck så måste det först gå att skapa ett tomt uttryck med defaultkonstruktor.

När du har klassen gränssnitt klart för dig, skriv in det i en headerfil `expression.h`. Tänk extra på att använda `const&` på parametrar av sammansatt typ(klasstyp) samt `const` efter medlemsfunktioner som inte ska modifiera objektet. Implementera sedan klassen i `expression.cc`. Allt utom konstruktorn ska vara trivialt att implementera tack vare nod-klasserna du redan skapat.

Konstruktorn behöver omvandla en sträng med ett postfix[4] till ett uttryckskträd. För att förenkla detta kräver vi att alla operander och operatorer skiljs åt av minst ett blanksteg. Vi tillåter inte heller några negativa tal. Negativa tal kan ändå skapas genom att subtrahera från 0. Algoritmen för att bygga upp ett uttryckskträd använder en stack. En stack kan du tänka på som en hög där vi antingen lägger något överst, eller tar bort det som ligger överst. Algoritmen för att bygga uttryckskträdet är som följer:

1. Läs in ett ord
2. Är ordet en giltig operand?
   (a) Skapa en ny operandnod av rätt typ
   (b) Lägg pekaren till den nya operandnoden på stacken
3. Är ordet en giltig operator?
   (a) Hämta och ta bort höger undernod från stacken
   (b) Hämta och ta bort vänster undernod från stacken
   (c) Skapa en ny operatornod av rätt typ med vänster och höger undernoder
   (d) Lägg pekaren till den nya operatornoden på stacken
4. Upprepa tills det inte finns fler ord
5. Roten till hel uttryckskträdet ligger nu överst på stacken

Uppgift 10 Utför algoritmen för hand

Utför algoritmen på några uttryck och rita upp hur det fungerar innan du börja koda. Du hittar exempel i Exempel-avsnittet.

Uppgift 11 Konstruktor för uttryck

Implementera algoritmen. Använd en `std::istringstream` och formaterad inmatning (`>>`) till sträng för att läsa in ord. Som stack ska du använda en `std::stack<Node*>` (där `Node*` är pekare till din nod-basklass). Funktionen ska generera ett `std::logic_error` för fyra enkla fall av fel i inmatat uttryck:

- Det saknas operand(er) till en operator
- Det finns mer än en nod kvar på stacken när orden är slut (saknad operator)
- Det finns ingen nod på stacken när orden är slut (tomt uttryck)
- Inmatat ord är varken operand eller giltig operator

[4]Se Exempel-avsnittet för att bekanta dig mer med postfix.

Tips: Avgöra vilken typ av operand ett ord är

```
#include <cctype>
#include <algorithm>

// Nedan är långt från ``foolproof'', men du behöver inte göra
// mer kontroller än detta för att skilja ut olika operander.
if ( std::all_of( begin(word), end(word), ::isdigit ) )
{
    // Vi har hittat ett heltal
}
else if ( isdigit(word.at(0)) )
{
    // Vi räknar ordet som flyttal
}
else if ( isalpha(word.at(0)) )
{
    // Vi räknar ordet som variabelnamn
}
else
{
    // Vi räknar ordet som en operator
}
```

Kontroll

Kontrollera att allt fungerar som tänkt. Utöver ovan fyra fel är det vanligt att en operator som ej är kommutativ råkar skapas fel, och det är lätt hänt att typkonverteringar blir fel eller ofullständiga. Om inmatat postfix inte stämmer överens med utskrivet postfix är det lätt att dra slutsatsen att något blivit fel.

Uppgift 12 Given kod för konvertering från infix

Det är lite jobbigt att mata in uttryck som postfix. Det är ju inte så vi brukar tänka. Det är även jobbigt att behöva mata in blanksteg runt varje operator. En kollega har löst problemen med en klass `Postfix` som skapar ett postfix från en given infix-sträng. I postfix-klassen sker dessutom inläsningen till en token istället för till en vanlig sträng. En token är en sträng med en inmatningsoperator (`operator>>`) som känner till att operatorer separerar operander även utan blanksteg. Använd den givna klassen `Postfix` så att dina uttryck kan konstrueras från infix istället.

Uppgift 13 Kommandon i huvudprogrammet

Skriv ett huvudprogram utgående från exemplet i inledningen. Se till att huvudprogrammet håller reda på aktivt uttryck och lägg till kommandon i huvudprogrammet. Om inmatad rad börjar med ett kolon (:) ska det tolkas som ett kommando. Annars ska raden tolkas som ett infix-uttryck som ersätter aktivt uttryck. Lägg till kommandona `:prefix`, `:infix`, `:postfix` och `:calc` för att skriva ut och

beräkna aktivt uttryck. Kommandot `:quit` eller `:exit` avslutar programmet. Se till att programmet aldrig kraschar. Om ett fel(undantag) inträffar ska felmeddelande skrivas ut och programmet fortsätta.

Bonus: Användning av git

Har du glömt checka in uppgifterna i git igen? Det är enklare att göra det löpande, men du vet sedan steg 2 hur du ska göra för att lösa det. Skapa sedan var sin branch döpt efter liuid i ert repository. Checka in var sitt nytt testfall i er respektive branch. Merga sedan er respektive branch med master.

Steg 4: Speciella medlemsfunktioner och minneshantering

I uppgifterna i steg 4 *ska* ska du utöver det du lärt dig i tidigare laborationer och uppgifter använda dig av:

- Förståelse av dynamiskt minne (minnesheapen)
- Destruktorer
- `delete` för att avallokera minne
- `valgrind --leak-check=full` för att söka minnesläckor

Inledning: Sökning efter minnesläckor

Nu har vi en riktigt bra klass för att hantera uttryck. Tyvärr läcker den minne som ett såll. Vi allokerar minne med `new` på många ställen utan att någonsin avallokera minnet med `delete`. Starta ditt program med `valgrind --leak-check=full ./a.out` och mata in några uttryck innan du avslutar programmet. När programmet avslutar ser du att Valgrind rapporterar hur mycket minne som inte avallokerats korrekt. Målet är att inte läcka något minne och inte ha några minnesfel. Tänk på att valgrind bara hittar minnesläckor i kod som faktiskt exekveras - det gäller alltså att programkörningen måste använda all kod som skrivits för att alla minnesfel ska upptäckas.

Uppgift 14 Kopiering och Flytt

I detta uppgift behöver du inte implementera alla fem speciella medlemsfunktioner. Vi väljer att förbjuda att kompilatorn använder kopieringskonstruktorn och kopieringtilldelningsoperatorn på samma sätt som vi gjort tidigare för Node-klasserna:

```
Expression(Expression const&) = delete;
Expression& operator=(Expression const&) = delete;
```

För att vårt huvudprogram ska fungera behöver vi dock implementera flyttkonstruktorn, flyttilldelningsoperatorn och tänka igenom i vilka lägen det är de bortagna kopierings-funktionerna som anropas och i vilka lägen det är flytt-versionerna som anropas.

Flyttkonstruktor

Flyttkonstruktorn skapar ett nytt objekt som tar över ägarskap av ett annat uttryck. Det andra uttrycket lämnas tomt. Kompilatorn anropar automatisk flyttkonstruktorn istället för kopieringskonstruktorn i situationer där det andra uttrycket inte används mer. I denna situation är det mycket effektivare att låta bli göra en djup kopia.

Flyttilldelning

Flyttilldelningsopertatorn ersätter ett befintligt objekt med ett annat uttryck. Kompilatorn anropar automatisk flytt-tilldelning istället för kopieringstilldelning i situationer där det andra uttrycket inte används mer. I denna situation är det mycket effektivare att låta bli göra en djup kopia. Tänk till hur du ska göra för att befintliga uttrycket inte ska läcka minne och hur du hindrar destruktorn att ta hand om det andra uttrycket du tar över. Det finns minst två lösningar.

Kopiering eller flytt?

Tids nog kommer du att hamna i situationen att ett uttryck behöver ersättas med ett annat. Situationen illustreras med kodalternativ (1) och kodalternativ (2). Vad är egentligen skillnaden mellan de båda alternativen?

```
Expression active{};

// (1)
Expression e{infix};
active = e;

// (2)
active = Expression{infix};
```

Tänk igenom vilka uttrycks-objekt som skapas med respektive alternativ och om vi kan säga något om hur dessa objekt kan tänkas användas senare i koden.

Kontroll

Normalt anropar du aldrig flytt-funktionerna själv. Kompilatorn hittar automatisk de situationer där de ska användas. Men vi behöver fortfarande kunna anropa dem i vår testning för att vara säkra på att de fungerar. Till detta kan du använda `std::move`.

Uppgift 15 Destruktor

Destruktorn körs automatiskt när objekt går ur scope eller avallokeras. Det ger oss möjlighet att avallokera pekare objektet håller reda på. Lägg till destruktorer för dina nodtyper där det är relevant. Tänk på att avallokering av en pekare av basklasstyp ska fungera korrekt oavsett vilken härledd nod som egentligen pekas ut. Destruktorn behöver ha dynamisk bindning för att det ska fungera.

Kontroll

Du ska ha kommit fram till en tom standard-destruktor med dynamisk bindning och en implementerad destruktor i härledd klass.

Bonus: Användning av git

Först en påminnelse om att checka in alla uppgifter som är klara. Se nu till att var och en har en uppdaterad lokal version av ert repository. Gör nu var för sig olika ändringar på samma ställe i samma fil. Checka in ändringen var för sig. Nu gör en av er `git push` och sedan gör den andre `git pull`. Det ska nu uppstå en merge-konflikt. Lös den tillsammans och skriv ned hur ni gjorde.

Bonus: Smartpekare

Istället för att manuellt hålla reda på när minne ska avallokeras kan vi låta ett klassobjekt av typen `std::unique_ptr<Node>` äga varje allokerad pekare. När klassobjektet går ur scope aktiveras dess destruktor som avallokerar den ägda pekaren. När klassobjektet tilldelas förs ägarskapat för pekaren över till den nya ägaren.

Modifiera din funktion som omvandlar en postfix-sträng till uttryckträd så den använder smartpekare.

- Stacken ska nu lagra smartpekare.
- För att allokera ett smartpekarobjekt används `std::make_unique<NodeType>(arg1, arg2)`.
- För att göra tilldelning mellan smartpekare används `smrtptr2 = std::move(smrtptr1)`.
- För att själv ta över ansvar för avallokering från en smartpekare används `Node* rwptr = smrtptr.release()`.

Kontroll smartpekare

När du lyckats med bonusuppgiften ska programmet inte längre läcka minne ens när noder råkar bli kvar på stacken till följd av felaktiga uttryck. Så fort funktionen avbryts kommer stackens destruktor aktiveras (det hände även tidigare). När stacken nu innehåller klassobjekt (av smartpekartyp) kommer destruktorn för varje objekt att anropas vilket i sin tur avallokerar minnet för den uttrycksnod smartpekarobjektet äger.

Bonus: Sparade uttryck

- Lägg till kommandot `:save` för att spara aktivt uttryck i en `std::vector<Expression>`.
- Lägg till kommandot `:list` för att lista alla sparade uttryck.
- Lägg till kommandot `:activate N` för att göra sparat uttryck N till aktivt uttryck.

Denna uppgift kräver att uttryck kan kopieras. Om du tänker till kan du ändå lösa uppgiften utan att implementera den frivilliga kopieringen nedan. Via de funktioner du har i `Expression` kan du skapa ett nytt objekt initierat till samma uttryck som ett befintligt objekt. Det nya objektet kan du sedan flytta dit det ska vara. Det är dock snyggare att implementera kopieringen. Båda varianterna godkänns. Däremot får du inte använda pekare för att undvika kopiering.

Bonus: Användning av git

När du är klar och checkat in alla uppgifter för din färdiga kalkylator ger du din assistent läsrättighet i ditt git-repository och sedan mailar du din assistent enligt följande:

```
Ämnesrad: TDDC76 git bonus, [liuid], [liuid]

1. Vi bekräftar att vi är klara med git bonusuppgiften och vill få den bedömd.

2. Förklara hur ni löste merge-konflikten från steg 4.

3. Skriv in en länk till ert projekt i gitlab:
   https://gitlab.liu.se/[liuid]/[projektnamn]

4. Klistra in utdata från git kommando:
   git --no-pager log --all --graph --oneline --decorate
```

Frivilligt: Implementation av Kopiering

Konstruktion av likadan nod

För att hantera kopiering av uttryckträd enkelt behöver vi några hjälpfunktioner. Börja med att skapa en funktion `construct` för varje operator. Den ska ta två nod-pekare som parametrar. Allt den ska göra är att skapa och returnera en ny nod av sin egen typ, med angivna argument som undernoder.

Klona uttryckträd

Skapa nu en funktion `clone` för varje nod. Den ska inte ta några parametrar. Den ska returnera en ny nod som är en djup kopia av noden den anropas på. Med djup kopia menar vi att även undernoderna ska vara djupa kopior. Resultatet av `clone` ska bli en exakt och helt självständig kopia av noden och dess undernoder. Med detta rekursiva angreppssätt är implementationen en rad kod. Rita upp ett uttryckträd, utför funktionen steg för steg, och försäkra dig om att du förstår hur det fungerar. Se Exempel-avsnittet för exempel.

Kontroll

Har du mer än tre implementationer av `clone`? Då har du onödig kodupprepning. Fundera ut hur du kan sammanföra snarlika implementationer till gemensam basklass. När vi nu har ett enkelt sätt att skapa en djup kopia av ett helt uttryckträd är det dags att implementera de speciella medlemsfunktionerna i uttrycksklassen.

Kopieringskonstruktor

Skapar ett nytt uttryck som ska vara en exakt djup kopia ett annat uttryck. Kompilatorn lägger ofta till anropa av kopieringskonstruktorn när objekt behöver kopieras. Om kopieringskonstruktorn saknas kommer du att få grunda kopior som orsakar dubbel avallokering när du gjort en korrekt destruktor.

Kopieringstilldelning

Ersätter ett befintligt uttryck med en exakt djup kopia ett annat uttryck. Om operatorn för kopieringstilldelning saknas kommer du att få grunda kopior när du använder `=`. Det orsakar dubbel avallokering när du gjort en korrekt destruktor. Implementera denna enligt principen kopiera och byt. Då skapar du först en djup kopia av argumentet med hjälp av kopieringskonstruktorn. Sedan låter du kopians datamedlemmar byta plats med befintliga uttryckets datamedlemmar. När sedan kopians objekt går ur scope tar dess destruktor hand om det som tidigare var befintliga uttrycket. Rita upp hur det fungerar!

Kontroll

När du har en korrekt destruktor tillsammans med korrekt kopieringskonstruktor och kopieringstilldelning ska dina uttryck kunna kopieras och tilldelas utan att några minnesläckor uppstår. Alla tre

funktioner måste vara korrekta för att allt ska fungera.

Observera: Om du kör ett catch testprogram genom valgrind måste du se till att alla allokeringar som sker i testprogrammet avallokeras.
Observera: När du matar in felaktiga uttryck kan det råka bli noder kvar på stacken. Detta ska inte läcka minne, men vi sparar åtgärd av det till bonusuppgiften smartpekare. Det räcker att ditt program inte läcker minne för korrekta inmatningar.

```
Expression e{"( 5 + 5 ) ^ 2 - 3.142 * 5 ^ 2 / 4.000"};
Expression f{"100 - 3.142 * 25 / 4.000"};
Expression g{ std::move(e) }; // Flyttkonstruktor
f = std::move(g); // Flytt-tilldelning
```

Frivilligt steg 5: Variabler i uttryck

I uppgifterna i steg 5 *ska* ska du utöver det du lärt dig i tidigare laborationer och uppgifter använda dig av:

- Förståelse av variabelscope (funktionsstack)
- Referens som datamedlem
- Överföring av referensparameter i flera steg (undvikande av kopior)
- `dynamic_cast<Requested_Type*>(object)` för att undersöka om ett objekt är av efterfrågad typ

Inledning

Titta tillbaka på uttrycket du gjorde objektorienterad analys på i steg 1. Där förekommer variabler (area och r) och en tilldelningsoperator (=). Nu ska vi lägga till detta i våra uttryck. Föra att göra det lite enklare börjar vi med att bara tillåta en variabel. Var ska den lagras? Om den lagras i variabelnoden så kommer det finnas flera kopior av variabeln värde och det är inte hållbart - det ger alltför många problem med att uppdatera och läsa av värdet, vilken kopia är "rätt"? Liknande problem får vi om värdet lagras i uttrycksklassen - vad händer om vi i nästa uttrycksobjekt vill använda samma variabel? Lösningen blir att den som använder uttryck själv får hålla reda på var variabler lagras och tillhandahålla en referens till lagringsutrymmet när uttryck skapas. Du ska använda detta angrepssätt. Vi tittar på hur ett huvudprogram kan se ut:

```
#include <iostream>
#include "expression.h"

using namespace std;

int main()
{
    string line;
    double storage{};

    while ( getline(cin, line) )
    {
        Expression e{line, storage};

        cout << e.evaluate() << endl;
    }
    return 0;
}
```

När programmet körs ska det fungera att räkna upp en variabel enligt följande exempel:

```
r = r + 1
1
r = r + 1
```

```
2
r = r + 1
3
```

Frivillig uppgift 1 Variabeltilldelning

Lägg till nodklasser för variabler och tilldelning. Tänk på att variabler *enbart* ska få nytt värde när tilldelningsoperatorn beräknas. Tänk på att funktioner för att läsa av en variabel eller uppdatera dess värde *enbart* är relevant för variabelnoden. Tänk på att så fort referenstecken (`&`) saknas när argument överförs till parameter eller datamedlem så sker en *kopiering*. Och så fort en kopiering sker står du med flera olika lagringsplatser för samma variabelvärde - med alla problem det medför. En tilldelningsoperator måste ha en variabelnod som vänster operand för att uttrycket ska vara giltigt. Om detta inte är uppfyllt ska beräkningen av tilldelningen generera ett `std::logic_error`.

Frivillig uppgift 2 Delberäkning

Du har nu kommit till den sista delen av labben. För att lösa den här uppgiften krävs att du förstått allt det du gjort fram till nu. Betrakta följande två uttryck:

```
Expression e{"1 + 2 * 3 - x + 4 * 5"};
Expression f{"7.000 - x + 20.000"};
```

Ser du att det är samma uttryck? Skillnaden är att vi delberäknat det första uttrycket så långt som möjligt i det andra uttrycket. Ser vi detta utifrån varje nods perspektiv blir det lite enklare. En operand delberäknas till sig själv. En operator kan delberäknas om dess båda noder är konstanter (heltalsnod eller flyttalsnod). Observera: vänster och höger nod kan *bli* konstanter efter att de delberäknats, så de måste delberäknas först. Tilldelningsnoder ska egentligen bara uppdatera sin variabel när de utvärderas - inte vid delberäkning. För att förenkla implementationen bortser vi från det. Lägg till kommandot `:fold` i huvudprogrammet för att utföra en delberäkning på aktuellt uttryck.

Kontroll

Lägg till testfall för att kontrollera att delberäkningen fungerar. Observera att en del uttryck som ser ut kunna delberäknas inte kan delberäknas beroende på uttryckskträdets uppbyggnad. När variabelnoden hamnar i en inre parentes hindrar den att alla yttre parenteser beräknas.

```
Infix:                 1 + 1  + 1  + 1  + 1  + 1  + 1  + 1  + x
Med parenteser: (((((((1 + 1) + 1) + 1) + 1) + 1) + 1) + 1) + x)
Delberäkning:          8 + x

Infix:                 x + 1  + 1  + 1  + 1  + 1  + 1  + 1  + 1
Med parenteser: (((((((x + 1) + 1) + 1) + 1) + 1) + 1) + 1) + 1)
Delberäkning:   (((((((x + 1) + 1) + 1) + 1) + 1) + 1) + 1) + 1)
```

Frivillig uppgift 3 Variabeltabell

Uppdatera din kod så att uttryck kan hantera godtyckligt många variabler. Det räcker nu inte att ditt huvudprogram har lagringsplats för en `double`. Använd istället en `std::map<std::string, double>` för att skapa en variabeltabell. Lägg till kommandot `:disp` för att skriva ut hela variabeltabellen.

Frivillig uppgift 4

Vill du gå mer bananas? Prova lägga till upprepning (£) där högernoden beräknas så länge vänsternoden är nollskild. Du kan även prova fakultetsoperatorn (!) vilket kan var lite klurigare eftersom det inte finns någon högeroperand.

Exempel

Vad är ett postfix?

Ett postfix är ett uttryck där operatorer skrivs efter sina båda operander och all information kommer i precis den ordning den behövs för att beräknas. Vi behöver inte veta vad som kommer längre fram i uttryck, bara vad vi redan stött på. Dessa egenskaper gör postfix enkla att hantera. Följande exempel är bara för att du ska få en uppfattning om postfix. Vi kommer inte använda precis dessa algoritmer.

```
Postfix: 1 2 + 3 * 4 -
Konvertera till infix:
Läs in 1, operand! spara "1" på stack
Läs in 2, operand! spara "2" på stack
Las in +, operator!
   hämta operander A och B från stack
   spara (A+B) på stack: "(1+2)"
Läs in 3, operand! spara "3" på stack
Las in *, operator!
   hämta operander A och B från stack
   spara (A+B) på stack: "((1+2)*3)"
Läs in 4, operand! spara "4" på stack
Läs in -, operator!
   hämta operander A och B från stack
   spara (A+B) på stack: "(((1+2)*3)-4)"
Ingen ytterligare indata, skriv ut resultat från stack
Klart!
```

```
Postfix: 1 2 + 3 * 4 -
Beräkning:
Läs in 1, operand! spara 1 på stack
Läs in 2, operand! spara 1 på stack
Las in +, operator!
   beräkna med två operander från stack
   spara resultat 3 på stack
Läs in 3, operand! spara 3 på stack
Läs in *, operator!
   beräkna med två operander från stack
   spara resultat 9 på stack
Läs in 4, operand! spara 4 på stack
Läs in -, operator!
   beräkna med två operander från stack
   spara resultat 5 på stack
Ingen ytterligare indata, skriv ut resultat från stack
Klart!
```

Konvertering av uttryckskträd till postfix

För att skapa ett postfix från ett uttryckskträd bestämmer vi oss för att varje nod ska ha en funktion som gör om den noden till ett postfix. Sedan anropar vi den funktionen för rot-noden. Nu är det bara att implementera funktionen. Men eftersom vi redan bestämt att den finns och fungerar i varje nod blir det enkelt. En operatornod ber först vänster barn om dess postfix, sedan höger barn om dess postfix och lägger sist till sin operatorsymbol. En operand-nod är ännu enklare, det räcker konvertera den till sträng. Klart. Vi har använt abstraktion med rekursion. Se även avsnittet Besöksordning av noder i uttryckskträd vid rekursiva operationer.

Beräkning av uttryckskträd

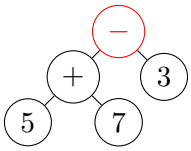
För att beräkna ett uttryckskträd bestämmer vi oss för att varje nod ska ha en funktion som gör om den noden till dess beräknade värde. Sedan anropar vi den funktionen för rot-noden. Nu är det bara att implementera funktionen. Men eftersom vi redan bestämt att den finns och fungerar i varje nod blir det enkelt. En operatornod ber först vänster barn om dess beräknade värde, sedan höger barn om dess beräknade värde och utför till sist sin operator på de två värdena. En operand-nod är ännu enklare, det räcker ge tillbaka dess värde. Klart. Vi har använt abstraktion med rekursion. Se även avsnittet Besöksordning av noder i uttryckskträd vid rekursiva operationer.

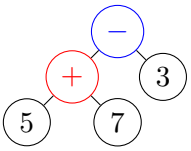
Djup kopiering av uttryckskträd

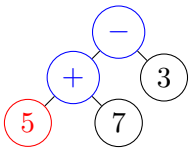
För att kopiera ett uttryckskträd bestämmer vi oss för att varje nod ska ha en funktion som gör en kopia av noden och alla dess undernoder. Sedan anropar vi den funktionen för rot-noden. Nu är det bara att implementera funktionen. Men eftersom vi redan bestämt att den finns och fungerar i varje nod blir det enkelt. En operatornod ber först vänster barn om dess kopia, sedan höger barn om dess kopia och sedan skapar den en kopia av sig själv med de två barnkopiorna som egna kopians barn. En operand-nod är ännu enklare, det räcker den ger tillbaka en kopia av sig själv. Klart. Vi har använt abstraktion med rekursion. Se även avsnittet Besöksordning av noder i uttryckskträd vid rekursiva operationer.

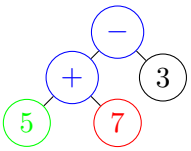
Besöksordning av noder i uttryckskträd vid rekursiva operationer

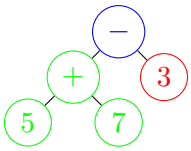
Vi kikar på i vilken ordning noderna besöks för ett exempel. Besöksordningen blir densamma oavsett vad (postfix, beräkning, kopia) vi vill utföra i varje nod. Rött symboliserar den noden som besöks i respektive steg, blått symboliserar en besökt nod som delegerat arbete till underliggande nod och som inte är helt klar ännu (den fortsätter sitt arbete när underliggande nod är klar) och grönt symboliserar att noden är färdigbehandlad. Notera att noden kan utföra uppgifter när den blir röd, när vi kommer tillbaka till blå nod och just innan den blir grön. Undersök för hand vad du får ut om du gör en utskrift i respektive läge.

(1) 

(2) 

(3) 

(4) 

(5) 

(6)