LINKÖPINGS UNIVERSITET
Institutionen för datavetenskap
Avdelningen för programvara och system (SaS)
Tommy Olsson 2014-02-07

TDDI14 Objektorienterad programmering

Laboration Kalkylatorn

Introduktion

I den här uppgiften att du ska arbeta med ett program för att behandla enkla aritmetiska uttryck. Delar av programmet är givet i form av klasser, funktioner och ett huvudprogram, en del fungerande, en del i form av kodskelett. Din uppgift är att komplettera och modifiera. Typiska objektorienterade konstruktioner kommer att användas, till exempel klasser, härledning/arv, polymorfi, dynamisk typkontroll (**typeid**, **dynamic_cast**) och dynamisk typomvandling (**dynamic_cast**).

Indata

Kalkylatorn ska kunna läsa antingen enbokstavskommandon eller infixuttryck. Exempel på infixuttryck:

```
1 + 2.5 * 3 - 4.7
x = 5 * 3 - (9 / 4) ^ 2
4711
(11147)
```

Ett uttryck måste innehålla minst en operand, annars betraktas det som ett tomt uttryck. Multipel tilldelning, som x = y = z = 0, ska inte tillåtas.

Operatorer

De operatorer som finns är =, +, -, *, / och ^ (exponentiering; operatorn ^ i C++ har en helt annan funktion). Samtliga ska vara binära operatorer med en semantik som överensstämmer med motsvarande operatorer i C++. Normala beräkningsregler gäller, dvs ^ har högst prioritet, därefter kommer * och /, sedan + och -, och lägst prioritet har =. Operatorerna ^ och = är högerassociativa, övriga vänsterassociativa. Associativiteten bestämmer beräkningsriktningen för sammansatta uttryck där ingående operatorer har lika prioritet:

`2+3-4`	*beräknas*	`(2+3)-4`	*vilket motsvarar postfixuttrycket*	`2 3 + 4 -`
`2^3^4`	*beräknas*	`2^(3^4)`	*vilket motsvarar postfixuttrycket*	`2 3 4 ^ ^`

Operander

En operand kan vara en variabel, ett icke-negativt heltal eller ett icke-negativt reellt tal. En variabels namn ska bestå av enbart små bokstäver. Ett heltal ska bestå av enbart decimala siffror (0-9), exempelvis 4711. Ett reellt tal ska bestå av minst en siffra, följt av en decimalpunkt, följt av minst en siffra, exempelvis 3.14. Ingen exponentdel förekommer i reella tal.

Variabler förutsätts endast förekomma till vänster om = i den obligatoriska uppgiften. Det medför att det inte behöver finnas stöd för att spara och återfinna värden för variabler som använts i tidigare uttryck. I den frivilliga bonusuppgiften ska en variabeltabell införas och då ska man kunna skriva uttryck som följande:

`x = 5`	*x ”definieras” (står till vänster om =)*
`y = x + 1`	*y ”definieras”, tidigare definierade x:s värde används*
`y + 2 * x - 1`	*y och x ska ha definierats tidigare, deras värden används här*

Då en variabel förekommer till vänster om = finns två alternativ:

1) har den inte har förekommit tidigare *definieras* den nu

2) har den definierats tidigare ska dess värde ändras

Om en variabel förekommer på annan plats än till vänster om =, måste variabeln vara definierad sedan tidigare och dess värde ska nu användas. Det innebär att det krävs stöd i programmet för att spara och återfinna variabler, deras namn och deras värde. För detta ska en klass Variable_Table konstrueras.

Utdata

Utdata från kalkylatorn ska vara olika former av de lagrade uttrycken. Ett uttrycks beräknade värde, dess postfixform och infixform samt trädutskrift ska kunna erhållas. Flera uttryck ska kunna lagras och visas. I den frivilliga bonusuppgiften ska även lagrade variabler och deras värden kunna visas.

Given programkod

Ett huvudprogram, en klass Calculator, hjälpfunktioner och testprogram en del annat är givet:

kalkylator	Innehåller main(), skapar ett Calculator-objekt och anropar dess run(), där det egentliga huvudprogrammet för kalkylatorn finns. (*komplett*) Fil: kalkylator.cc, Makefile
Calculator	Huvudklassen för kalkylatorn, har en medlemsfunktion run() där menyslingan för kalkylatorn körs: läs kommando – kontrollera – utför. (*i princip komplett, tillägg ska dock göras efter hand*) Filer: Calculator.h, Calculator.cc
Expression	Klass för att representera uttryck. Som intern representation för uttrycket i Expression ska ett träd av Expression_Tree-noder användas. En hjälpfunktion make_expression(), för att skapa Expression-objekt är given, samt stöd för infix-till-postfix-omvandling och omvandling från postfix till uttryckträd. (*skelett/stubbar*) Filer: Expression.h, Expression.cc
Expression_Tree	Basklass för klasser som ska representera elementen i uttrycken, dvs operander och operatorer. Varje slags element ska representeras av en specifik klass, till exempel ska + representeras av en klass Plus och heltal av en klass Integer. (*skelett/stubbar*) Filer: Expression_Tree.h, Expression_Tree.cc

Kommentarer skrivna med STORA bokstäver ska vara borttagna då du lämnas in dina lösningar.

Din uppgift

I din uppgift ingår att konstruera följande klasser; närmare beskrivning längre fram, se även **klassdiagrammet** på sista sidan.

- **Expression** ska representera uttryck och ha uttryckträd som intern representation. Detta innebär konstruktion av en enskild, icke-trivial klass.
- **Expression_Tree** med subklasser ska representerar de olika typerna av trädnoder som kan finnas i uttrycksträden. Detta innebär konstruktion av en icke-trivial, polymorf klasshierarki.
- **Undantagsklasser** ska definieras för de olika komponenterna och undantagshantering ska införas i programmet. Undantagsklasserna ska härledas från lämplig(a) undantagsklass(er) i exception-hierarkin.
- **Undantagshantering**. I de givna funktionerna i Expression.cc hanteras fel genom att ett felmeddelande skrivs ut på cerr, därefter termineras programmet genom anrop av funktionen exit(). Detta ska ersättas med att undantag kastas. Programmet kommer då att överleva fel och *minnesläckor* som eventuellt kan uppstå i dessa funktioner måste elimineras.
- **Minnesläckor ska elimineras.** Expression_Tree-objekt skapas dynamiskt och det innebär problem om minnestilldelningen misslyckas, med risk för minnesläckor.
- **Variable_Table** (frivillig) ska användas för att hantera variabler som införs i uttrycken och deras värden.

Var ska jag börja?

Klassdiagrammet sist i häftet visar hur klasserna är relaterade. Om vi bortser från Variable_Table:

- Expression_Tree-klasserna ska inte känna till (användas) någon av de andra klasserna
- Expression måste känna till Expression_Tree
- Calculator måste känna till Expression

Detta givet kan det vara lämpligt att börja med Expression_Tree.

- Kommentera bort en del av de givna klasskeletten – det kan ju räcka med någon/några operatornodklasser och en operandnodklass till att börja med. Övriga klasser kommer att vara snarlika.
- Börja med det som är nödvändigt för att kunna skapa ett enkelt uttryckträd och skriva ut det.
- *Ta lite i taget och testa ofta!*

Klassen Calculator

Calculator, som är given, är huvudklassen i programmet och utgör själva kalkylatorn.

Calculator har en publik medlemsfunktion run() som anropas för att starta kalkylatorn och run() kan sägas utgöra det egentliga huvudprogrammet – det är i run() som huvudslingan för att läsa in, kontrollera och utföra kommandon finns. När kalkylatorn körs kan det kan se ut enligt följande exempel, användarens inmatning markerad med **fet** stil:

```
Välkommen till Kalkylatorn!

H, ?  Skriv ut denna hjälpinformation
U     Mata in ett nytt uttryck
B     Beräkna uttrycket
P     Visa uttrycket som postfix
T     Visa uttrycket som ett träd
S     Avsluta kalkylatorn
>> U
x=1+2*4
>> P
x 1 2 4 * + =
>> T
      4
     /
    *
     \
      2
   /
  +
   \
    1
 /
=
 \
  x
>> B
9
>> S
Kalkylatorn avlutas, välkommen åter!
```

Hjälpinformation skrivs först ut och sedan uppmanas till inmatning med ledtexten "**>>**" på en ny rad. Då kan ett enbokstavskommando ges. Om kommandot U (eller u) ges kan man skriva en rad med ett infixuttryck. När man avslutar raden sker följande:

- den inmatade raden, som ska vara ett infixuttryck, läses in som en sträng
- strängen ges till funktionen make_expression() som returnerar ett objekt av typen Expression
- Expression-objektet lagras i kalylatorn som det *aktuella uttrycket*

Övriga kommandon, utom S, används för att operera på det aktuella uttrycket.

Kommandorepertoar

Endast det senast inmatade uttrycket ska (inledningsvis) lagras i kalkylatorn; när man matar in ett nytt uttryck ersätts det föregående. Följande kommandon finns:

```
H, ?  Skriv ut denna hjälpinformation
U     Mata in ett nytt uttryck
B     Beräkna uttrycket
P     Visa uttrycket som postfix
T     Visa uttrycket som ett träd
S     Avsluta kalkylatorn
```

Repertoaren ska utvidgas efter hand men först ska Expression_Tree och Expression konstrueras.

Klasshierarkin Expression_Tree

För att representera noderna i ett uttryckträd ska en polymorf klasshierarki konstrueras. Detta är en mycket viktig del av laborationen, stor vikt läggs vid att klasshierarkin och de ingående klasserna konstrueras väl!

- **Expression_Tree** ska vara en gemensam abstrakt basklass för klasshierarkin. Den ska definiera det gemensamma gränssnittet (operationerna) för alla trädnodklasser.
- Elementen i uttrycken kan delas upp i två huvudkategorier, *binära operatorer* och *operander*. Detta ska avspeglas i klasshierarkin av de abstrakta klasserna **Binary_Operator** och **Operand**.
- Varje *operator* ska representeras av en egen klass och varje klass ska ansvara för sina specifika uppgifter. Om evaluate() för en plusnod anropas ska den anropa evaluate() i sina två subträd, addera värdena som returneras och sedan returnera resultatet av additionen. Klasserna döps lämpligtvis till **Assign**, **Plus**, **Minus**, **Times**, **Divide** och **Power**.
- Varje typ av *operand*, dvs heltal, reellt tal eller variabel, ska representeras av en motsvarande klass. Klasserna döps lämpligtvis till **Integer**, **Real** och **Variable**.

Följande funktioner ska finnas för Expression_Tree-hierarkin:

- evaluate() ska beräkna värdet av det (del)uttryck som ett (del)träd representerar. För en tilldelningnod ska evaluate() först beräkna uttrycket i höger subträd, sätta värdet på variabelnod till vänster till detta värde och slutligen returnera värdet.
- get_postfix() ska returnera uttrycket i ett (del)träd i postfixform som en sträng.
- print(*utström*) ska göra en så kallad trädutskrift av ett (del)träd på en angiven utström. Det är tillåtet att göra tillägg i parameterlistan, utöver den utström som ska vara första parameter, för att stödja utskriften av trädet.
- str() ska returnera en nod representerad som en sträng. För operatornoder operatorsymbolen (”+”, ”-”, osv.), för talnoder deras värde på strängformat (exempelvis strängen ”37.5” för en realtalsnod som lagrar 37.5), för en variabel dess namn, inte värdet. Flera andra funktioner kan ha användning av str().
- clone() ska göra en kopia av ett (del)träd och returnera en pekare till kopian. Denna operation är mycket användbar för att implementera vissa andra operationer.

  clone() ska vara den enda publika möjligheten att kopiera Expression_Tree-objekt! clone() ska använda kopieringskonstruktorn för att göra själva kopieringen.

Variabelklassen ska, utöver de gemensamma funktionerna för alla trädnodklasserna även ha funktionerna:

- set_value() ändrar variabelns värde.
- get_value() returnerar variabelns värde; samma sak som evaluate() egentligen men finns en set-funktion förväntar man sig normalt en motsvarande get-funktion.

I programmet kommer det kanske inte att finnas något påtagligt behov av dessa funktioner men de bör finnas för en komplett variabelklass och variabelns värde ska lagras i Variable-objektet. Om en variabeltabell införs kan en annan lösning övervägas.

Speciella medlemsfunktioner som inte nämnts ovan ska deklareras på något sätt – ”defaultas” eller definieras om de ska kunna användas, ”deletas” om de inte ska kunna användas.

Felhantering i Expression_Tree

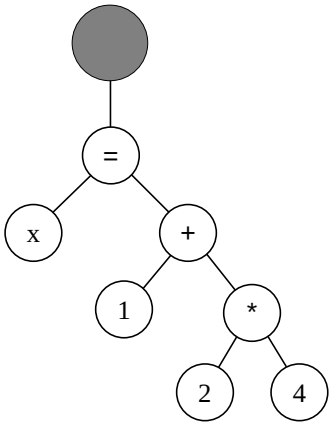
Fel ska hanteras med undantag, i analogi med övriga klassers felhantering.

Uttryckträden består av noder som tilldelas minne dynamiskt och det kan misslyckas. Om det inträffar ska detta hanteras på ett starkt undantagssäkert sätt.

Problem kan även uppstå i samband med att uttryck beräknas, exempelvis division med 0.

Klassen Expression

Klassen Expression ska representera uttryck i Calculator. Som intern representation ska Expression ha ett uttryckträd bestående av noder av typen Expression_Tree. Detta illustreras i figuren till höger, där den skuggade noden ska föreställa ett Expression-objekt, övriga noder är olika typer av Expression_Tree-objekt.



Expression ska ha följande publika operationer:

- evaluate() ska returnera värdet av uttrycket i uttryckträdet.
- get_postfix() ska returnera en sträng med uttrycket i postfixform.
- print_tree(*utström*) ska skriva ut uttryckträdet på angiven utström, i form av en så kallad trädutskrift (se nedan).
- empty() ska besvara om ett uttryckträd är tomt eller ej.
- swap() ska byta innehåll (träd) på två Expression-objekt.

Expression-objekt skapas av den givna funktionen make_expression().

Expression är en form av containerklass och ska ha de operationer som förväntas för en sådan. Kopiering och tilldelning ska vara tillåten.

Glöm inte att ersätta felutskrifterna/exit()-anropen i de givna hjälpfunktionerna med undantagskastning!

Felhantering i Expression

Om en operation på ett Expression-objekt inte kan utföras ska undantag kastas. Definiera en undantagsklass expression_error för detta. Använd någon lämplig klass i exception-hierarkin som basklass.

Kontrollera koden, både den givna och din egna, efter potentiella problem, exempelvis att undantag kan komma att kastas, och se till att hantera detta på ett *starkt undantagssäkert* sätt. Om bad_alloc kastas ska det hanteras *undantagsneutralt*, dvs det får passera vidare till anroparen av operationen utan annan åtgärd.

Trädutskrift

En trädutskrift innebär att ett uttryckträd ska skrivas ut på ett sätt som visar trädstrukturen. Att göra det riktigt snyggt är komplicerat och kräver grafik. Vi ska hålla oss till en betydligt enklare form av trädutskrift, som kan göras med enkel teknik i ett terminalfönster. Om vi har postfixuttrycket ”x 5.3 2 7.1 * + =” (som motsvarar infixuttrycket x = 5.3 + 2 * 7.1) ska följande träd skrivas ut:

```
        7.1
       /
      *
       \
        2
     /
   +
    \
     5.3
  /
=
  \
   x
```

Tilldelningsoperatom (=) sitter i rot-noden.

I vänster subträd finns endast noden x.

I höger subträd är + rotnod, 5.3 dess vänstra subträd och i höger subträd till + finns ett träd med * i roten, 2 till vänster och 7.1 till höger.

En del grenar (snedstreck) når inte fram till den ”nod” de borde men det får vi stå ut med. Det gäller grenen från = till +, och från + till * i detta fall.

```
Värde..: 19.5
Postfix: x 5.3 2 7.1 * + =
Infix..: x = 5.3 + (2 * 7.1)
```

Vrid pappret 90 grader medurs för att "se" trädet!

Utöka Calculator

Två tillägg i Calculator ska göras och kommandorepertoaren utökas i enlighet med detta. Nya operationer som krävs ska läggas till i konsekvens med övriga, dvs i Expression och Expression_tree.

Ett godtyckligt antal uttryck ska kunna lagras

Då ett nytt (korrekt) uttryck matas in ska det bli *aktuellt uttryck* och föregående aktuella uttryck ska sparas i en kronologiskt ordnad datastruktur, där alla tidigare inmatade uttryck som inte raderats ska finnas. Enklast hantering blir det om man sparar ett nytt korrekt uttryck i datastrukturen redan i samband med att det lästs in; vänta inte till nästa uttryck matas in. Välj en standardcontainer som lager för uttrycken. Det aktuella uttrycket ska alltid finnas i variabeln current_expression och i datastrukturen med sparade uttryck.

För att kunna referera till uttrycken behöver en del kommandon kunna ta ett argument, i form av ett heltal som anger ordningsnumret för ett uttryck. Kommandot B (beräkna uttryck), exempelvis, ska kunna ges både som B, beräkna det aktuella uttrycket, och som ’B *n*’, beräkna uttryck nummer *n*. Det kan internt i Calculator vara praktiskt att skilja mellan de kommandon som kan ta argument och de som inte kan.

Kommandot N ska skriva ut hur många uttryck som finns lagrade, ’A *n*’ ska göra uttryck nummer *n* till aktuellt uttryck (det för närvarande aktuella uttrycket ska redan vara lagrat), R ska radera aktuellt uttryck.

Kalkylatorn ska kunna visa uttryck som infix

Kalkylatorn ska kunna återskapa uttryck som de matades in. Detta är inte helt enkelt. När infix omvandlas till postfix försvinner alla parenteser och beräkningsordningen avspeglas i stället av elementens ordning i postfix-en. När ett uttryckträd genereras ur postfix avspeglar trädets struktur beräkningsordningen; delträd motsvarar deluttryck och beräkningsordningen avspeglas av delträdens inbördes ordning i trädet och deras djup. Exakt det uttryck som matades in kan man inte räkna med att återskapa, det kan ha funnits redundanta parenteser. Däremot kan man relativt lätt återskapa infixen med full parentetrisering av deluttryck. Följande kan vara en komplett parentetrisering av ett uttryck.

```
(x = (5.3 + (2 * 7.1)))
```

De två yttersta nivåerna är enkla att eliminera och det är kravet som ska uppfyllas, se nedan.

```
x = 5.3 + (2 * 7.1)
```

Lägg till kommandot I, visa ett uttryck som infix, och kommandot L, lista alla uttryck som infix. L ska ge en lista med alla uttryck, numrerade 1 och uppåt, se nedan.

```
1:  x = 5.3 + (2 * 7.1)
2:  y = 1
3:  z = x + y
```

Utökad kommandorepertoar

Ovanstående tillägg medför modifieringar och tillägg i kommandorepertoaren. Den kompletta kommando-repertoaren ska nu vara följande, nya kommandon markeras med *.

```
H, ?  Skriv ut denna hjälpinformation
A n   Gör uttryck n till aktuellt uttryck          *
B     Beräkna aktuellt uttryck
B n   Beräkna uttryck n                            *
I     Visa aktuellt uttryck som infix              *
I n   Visa uttryck n som infix                     *
L     Lista alla uttryck som infix                 *
N     Visa antal lagrade uttryck                   *
P     Visa aktuellt uttryck som postfix
P n   Visa uttryck n som postfix                   *
R     Radera aktuellt uttryck                      *
R n   Radera uttryck n                             *
T     Visa aktuellt uttryck som ett träd
T n   Visa uttryck n som ett träd                  *
U     Mata in nytt uttryck
S     Avsluta kalkylatorn
```

Klassen Variable_Table – frivillig bonusuppgift

Om lösningen av denna uppgift uppfyller kraven och dessutom samtliga deadline som gäller för kursens laborationsuppgifter hålls, ges *1 bonuspoäng* som kan tillgodoräknas för betyg 4 eller 5 vid de tre första tentamenstillfällena efter ordinarie deltagande i kursen (Vt2, augusti, och oktober 2013).

Klassen Variable_Table ska finnas i en egen mapp med namnet Variable_Table, vid sidan av de givna mapparna Calculator och Expression. Modifera Makefile i Calculator för att hantera detta på samma sätt som mappen Expression.

I variabeltabellen ska alla variablers namn och värde lagras. Följande funktioner ska finnas:

- insert(*namn, värde*) ska lägga till en ny variabel och dess värde i tabellen.
- remove(*namn*) ska ta bort en variabel och dess värde ur tabellen.
- find(*namn*) ska returnera **true** om variabeln finns i tabellen, annars **false**.
- set_value(*namn, värde*) ska ändra värdet för en variabel som finns i tabellen.
- get_value(*namn*) ska returnera värdet för en variabel som finns i tabellen.
- list(*utström*) ska skiva ut alla variabler i tabellen på en utström. För varje variabel skrivs först dess namn ut, följt av ett kolon och ett mellanrum och sist på raden variabelns värde.
- clear() ska tömma tabellen.
- empty() ska returnera **true** om tabellen är tom, annars **false**.

Använd en lämplig standardcontainer för att implementera variabeltabellen. Variable_Table går utmärkt att testa separat, skriv ett testprogram och en Makefile för att göra det i mappen Variable_Table.

Felhantering i Variable_Table

Ett Variable_Table-specifikt undantag variable_table_error ska kastas av operationer som förutsätter att en variabel måste finnas i tabellen för att de ska kunna utföras.

Kommandorepertoar då variabeltabell införts

När variabeltabellen lagts till ska kalylatorn även ha kommandona V och X. Den nu helt kompletta kommandorepertoaren visas nedan (de nya kommandona markeras med *).

```
H, ?  Skriv ut denna hjälpinformation
U     Mata in nytt uttryck
B     Beräkna aktuellt uttryck
B n   Beräkna uttryck n
I     Visa aktuellt uttryck som infix
I n   Visa uttryck n som infix
L     Lista alla uttryck som infix
P     Visa aktuellt uttryck som postfix
P n   Visa uttryck n som postfix
T     Visa aktuellt uttryck som ett träd
T n   Visa uttryck n som ett träd
N     Visa antal lagrade uttryck
A n   Gör uttryck n till aktuellt uttryck
R     Radera aktuellt uttryck
R n   Radera uttryck n
V     Lista alla variabler                    *
X     Radera alla variabler                   *
S     Avsluta kalkylatorn
```

Utskriften från kommandot V (Lista alla variabler) ska för varje variabel skriva ut dess namn, ett kolon, ett mellanrum och sist på raden variabelns värde, exempelvis om x har värde 4711 och y har värdet 11147.

```
x: 4711
y: 11147
```

Tänkvärt

Det är fullt medvetet att klasserna som ska konstrueras inte är specificerade i detalj. Det ingår i uppgiften att utifrån givna specifikationer och allmäna regler för konstruktion av klasser reflektera över klassernas detaljutformning och själv komma fram till bra lösningar, i alla avseenden.

Tänk noga igenom, för varje klass, vad som ska kunna göras med objekt av klassen ifråga och av vem. Till exempel ska ett program som ska *använda* uttryckträd bara behöva känna till klassen Expression men inte Expression_Tree, som är intern representation för uttryckträdet i klassen Expression.

Tänk noga igenom för en klasshierarki vad som är gemensamt för samtliga objekt, vad som är specifikt för en viss kategori av objekt och vad som är specifikt för varje specifik typ av objekt. Det är viktigt att en klasshierarki avspeglar de olika objektens gemensamma och specifika egenskaper på ett naturligt sätt. Inför datamedlemmar och (tillhörande) medlemsfunktioner i hierarkin där de logiskt hör hemma; datamedlemmar och medlemsfunktioner som endast används av vissa subklasser har hamnat fel!

Fel kan inträffa i de klasser som du ska konstruera, även om mycket fångas redan i samband med kontrollen av infixuttrycken. Fel i egna klasser ska hanteras med undantag, såvida problem inte kan lösas lokalt.

Annat viktigt att tänka på:

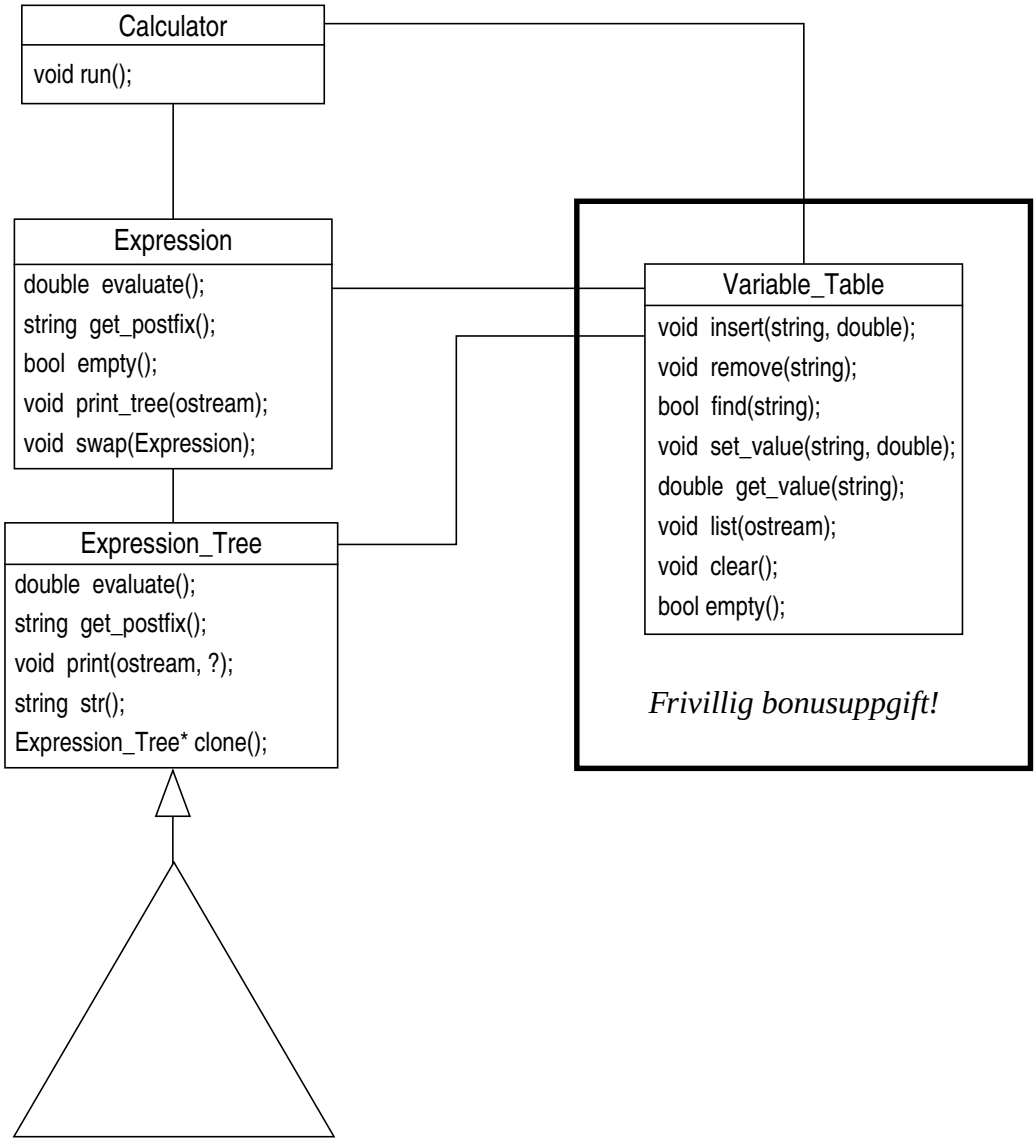
- åtkomstskydd för medlemmar i klasser
  - hur ska medlemmar delas upp mellan **public** och **private**, i fallet arv även **protected**, för att erhålla en bra kombination av skydd och åtkomst?
  - Kan **friend**-deklaration motiveras i vissa situationer?
- initiering och kopiering
  - vad ska vara möjligt/tillåtet för olika objekt och hur ska det fungera?
  - duger de kompilatorgenererade versionerna av konstruktorer, destruktor och tilldelningsoperatorn eller måste du definiera sådana själv?
  - har man kopieringskonstruktor och/eller kopieringstilldelningsoperator, ska man normalt också ha deras move-motsvarigheter.
- destruering
  - till exempel att säkerställa återlämning av dynamiskt minne.
- konsekvent användning av **const**-deklaration och referenser, i alla sammanhang.
- var noga med namngivning!

Arbeta **inte** *enligt ”big bang-metoden” – koda lite i taget och testa noga successivt!*

Kontrollera mot checklistor i stödmaterialet och kodningsstandarden High Integrity C++ Coding Standard!

Klassdiagram

Nedan finns ett klassdiagram som visar hur klasserna är relaterade genom association eller arv. Klasshierarkin för uttryckträdsnoderna visas bara med basklassen Expression_Tree och för övrigt med en triangel.



Angivna deklarationer för medlemsfunktionerna är ej de exakta med avseende på parametrars och returvärdens **cv**-kvalificering (**const**), referensegenskap, etc.

Expression ska även ha en swap() som inte är medlem och dessutom tillhör hjälpfunktionen make_expression() klassen Expression.

Det kan tänkas att fler medlemsfunktioner för internt bruk kan vara lämpligt att lägga till.