Laboration 3

Kalkylator för aritmetiska uttryck

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 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**).

Siktar du på VG? Följ planen så att du hinner att bli klar till deadline! Det ska inte vara några problem att först lösa uppgiften för G och sedan lägga till det som krävs för VG.

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)
x
```

Ett uttryck måste innehålla minst en operand, annars betraktas det som ett tomt uttryck. Multipel tilldelning, som x = y = z = 0, ska inte tillåtas (enkelt tillägg i den givna koden kan göras).

Operatorer

Operatorerna är =, +, -, *, / och ^ (exponentiering; operatorn ^ i C++ har en helt annan funktion). Samtliga ska vara tvåställiga (binära) operatorer med 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.

Variabelhantering för betyg G. Variabler förutsätts endast förekomma till vänster om =. Det medför att det inte behöver finnas stöd för att spara och återfinna värden för variabler som använts i tidigare uttryck.

Variabelhantering för betyg VG. Variabler ska kunna förekomma på godtycklig plats i uttryck, som exempelvis i följande uttryck:

`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; 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. Detta 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. För G ska ett uttrycks värde, postfixform och trädutskrift kunna erhållas. För VG ska flera uttryck kunna lagras och utöver vad som krävs för G ska även infixformen kunna erhållas och variabler och deras värden ska kunna visas.

Given programkod

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

kalkylator — Innehåller main(). Ett Calculator-objekt skapas och dess run-funktion anropas, där det egentliga huvudprogrammet för kalkylatorn finns. *Koden är komplett för G och VG.*
Fil: **kalkylator.cc**, **Makefile**

Calculator — Huvudklass för kalkylatorn. I medlemsfunktionen run() finns menyslingan för kalkylatorn: läs kommando – kontrollera – utför. *Koden är komplett för G men för VG krävs modifieringar.*
Filer: **Calculator.h** och **Calculator.cc**

Expression — Klass för att representera uttryck. Som intern representation för uttrycket ska ett länkat träd av Expression_Tree-noder användas. Den givna funktionen make_expression() skapar ett Expression-objekt med hjälp av interna funktioner för omvandling av infix till postfix och omvandling från postfix till uttryckträd. *Kodskelett givet – modifiera och komplettera.*
Filer: **Expression.h** och **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. *Kodskelett givet – modifiera och komplettera.*
Filer: **Expression_Tree.h** och **Expression_Tree.cc**.

Din uppgift

I uppgift en ingår att konstruera följande klasser. Närmare beskrivning kommer längre fram, se även klassdiagrammet på sista sidan.

- **Expression** ska representera uttryck och använda 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 ett uttryckträd. *Detta innebär konstruktion av en icke-trivial, polymorf klasshierarki.*
- **Undantagsklasser** ska definieras för de olika komponenterna och undantagshantering ska införas i programmet. Härled från lämplig undantagsklass i exception-hierarkin.
- I de givna funktionerna make_postfix() och make_expression_tree() i Expression.cc hanteras fel genom att ett felmeddelande skrivs ut på cerr, varefter programmet avslutas 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 ska därför elimineras.

För VG

En klass **Variable_Table** för att lagra variablers namn och värde och en tillhörande undantagsklass ska konstrueras. Använd filerna **Variable_Table.h** och **Variable_Table.cc**.

Kommandorepertoaren i Calculator ska utökas, vilket kräver tillägg i Calculator.

Konstruktion av klasser och felhantering är *extra* viktigt för VG.

Expression_Tree-objekt skapas dynamiskt i samband med exempelvis kopiering, vilket innebär problem om minnestilldelningen misslyckas. Sådana minnesläckor ska elimineras.

Var ska du börja?

Klassdiagrammet sist i häftet visar delvis hur klasserna är relaterade. Vad som inte framgår är riktningen på relationerna:

- Expression_Tree-klasserna behöver (ska) inte känns till (använda) 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.

- Det kan vara lämpligt att kommentera bort en del av de givna klasskeletten – det kan ju räcka med någon eller några operatornodklasser och en operandnodklass.
- Börja med sådant som är nödvändigt för att kunna skapa ett enkelt uttryckträd och skriva ut det.
- Ta lite i taget och testa!

Det ska inte innebära några problem att först lösa för G och sedan lägga till för VG.

Klassen Calculator

Calculator är huvudklassen i programmet och utgör själva kalkylatorn. Den är given för nivå G.

Calculator har en publik medlemsfunktion run() som anropas för att starta kalkylatorn och det är i run() som huvudslingan för att läsa in, kontrollera och utföra kommandon finns. När kalkylatorn körs kan det kan se ut som i exemplet nedan, användarens inmatning är 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 och sedan slå tangenten ENTER. Då sker följande:

- den inmatade raden, som alltså 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*, som andra kommandon sedan utförs på

Kommandorepertoar

Den givna Calculator-klassen har en kommanorepertoar och funktionalitet som motsvarar nivå G. Endast det senast inmatade uttrycket lagras i kalkylatorn; när man matar in ett nytt uttryck ersätts föregående.

Följande nivå-G-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 för VG men även om du siktar på det är det lämpligt att först konstruera Expression_Tree och Expression för G-nivå.

Tillägg i Calculator för VG

För VG ska kommandorepertoaren ska utvidgas enligt nedan. Vissa kommandon ska kunna ta ett argument.

```
H, ?  Skriv ut denna hjälpinformation
U     Mata in nytt uttryck
B     Beräkna aktuellt uttryck
B n   Beräkna uttryck n
P     Visa aktuellt uttryck som postfix
P n   Visa uttryck n som postfix
I     Visa aktuellt uttryck som infix
I n   Visa uttryck n som infix
L     Lista alla uttryck som infix
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
```

Nya operationer som krävs ska läggas till i konsekvens med övriga, dvs i Expression och Expression_Tree.

Ett godtyckligt antal uttryck kunna lagras i kalkylatorn. Då ett nytt, korrekt uttryck matas in ska det bli det *aktuella uttrycket* och föregående aktuella uttryck ska sparas en kronologiskt ordnad lista där alla tidigare inmatade uttryck (som inta har raderats) sparas. Det kan vara praktiskt att spara ett nytt uttryck i uttryckslistan redan i samband med att det läses in, dvs att inte vänta till dess nästa uttryck matas in.

Argumentet `n` som förekommer i vissa kommandon ovan är ordningsnumret för ett lagrat uttryck. `n` ska gå från 1 och uppåt. Utskriften från kommandot `L` (Lista alla uttryck som infix) ska visa ordningsnumret för varje uttryck först på raden, enligt följande (det finns tre sparade uttryck):

```
1: x = 1 + (2 * 4)
2: x / 2 - 1
3: y = ((x ^ 2) + (3 * x)) - 5
```

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. I exemplet nedan har x värde 9 och y värdet 103:

```
x: 9
y: 103
```

Variabler ska lagras i en variabeltabell och för det ska en klass Variable_Table konstrueras. Uttrycken 1 och 2 ovan är exempel på uttryck som definierar variabler, x respektive y. Uttryck 1 ska medföra att x och dess värde (värdet av högerledet till =) sparas i variabeltabellen. Uttryck 3, där x används för att beräkna värdet, ska lägga till y och dess värde i variabeltabellen.

Klasshierarkin Expression_Tree

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

- **Expression_Tree** ska vara en gemensam abstrakt polymorf basklass för klasshierarkin. Den ska definiera det gemensamma gränssnittet 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.
- Varje typ av *operand*, dvs heltal, reellt tal eller variabel, ska representeras av en motsvarande klass.

Följande funktioner ska finnas för Expression_Tree-hierarkin på nivå G:

- 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(ostream) ska göra en s.k. 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 …
- 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 att dess namn, inte värdet. Flera andra funktioner kan använda 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.

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() men set-get är ett par).

Felhantering i Expression_Tree

Fel som kan uppstå i samband med att uttryck beräknas och ytterligare fel kan uppstå i VG-versionen. Felen ska hanteras med undantag, i analogi med övriga klassers felhantering.

Tillägg i Expression_Tree för VG

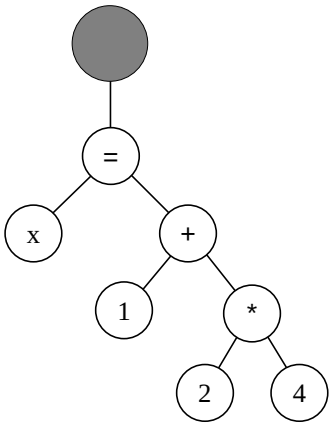
Funktionen get_infix() ska returnera uttrycket i ett (del)träd i infixform.

Uttryckträden består av noder som tilldelas minne dynamiskt och det kan misslyckas. Om det inträffar ska detta hanteras så att programmet inte läcker minne.

Beräkning av ett uttrycks värde ska göras av funktionen evaluate() i Expression som i sin tur delegerar uppgiften till evaluate() i Expression_Tree, där den egentliga beräkningen görs. Konsekvensen av detta är att dessa funktioner måste känna till variabeltabellen, dels för att kunna ta reda på en variabels värde när den ingår i beräkningen av ett uttryck, dels för att kunna lägga till en variabel i tabellen när den ”definieras”. Expression och Expression_Tree behöver alltså ha tillgång till den variabeltabell som Calculator använder. Detta bör diskuteras med assistenten innan kodningen.

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 olika typer av Expression_Tree-objekt.



Expression ska ha följande publika operationer för nivå G:

- evaluate() ska returnera värdet av uttrycket i uttryckträdet.
- get_postfix() ska returnera en sträng med uttrycket i postfixform.
- print_tree(ostream) ska skriva ut uttryckträdet på angiven utström, i form av en s.k. trädutskrift (se nedan).
- empty() ska besvara om ett uttryckträd är tomt eller ej.
- swap() ska byta innehåll (träd) på två Expression-objekt.
- de speciella medlemsfunktioner som kan anses lämpliga för objekt av typen Expression.

Expression-objekt skapas av den givna funktionen make_expression(). I make_expression() skapas ett uttryckträd med hjälp av dynamisk minnestilldelning (**new**) – risk för minnesläcka. make_expression() skulle eventuellt, med vissa fördelar, kunna göras om till en medlem av Expression.

make_expression() returnerar inga "tomma uttryck" (Expression-objekt där uttryckträdet är en tompekare, **nullptr**, men sådana skulle kunna skapas av funktioner som använder flyttsemantik.

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.

Minnesläckor som kan uppstå i funktionen make_expression_tree() då undantagshantering ersätter den givna felhanteringen ska hittas och elimineras.

Om bad_alloc kastas i en Expression-operation ska det hanteras *undantagsneutralt*, dvs det ska passera till anroparen av operationen och till exempel inte bytas ut mot ett annat slags undantag.

Tillägg för i Expression för VG

Funktionen get_infix() ska returnera en sträng med uttrycket i infixform.

Uttryckträdets struktur avspeglar beräkningsordningen för delutrycken i ett uttryck och när motsvarande infixuttryck ska återskapas kan parenteser behöva sättas in, men redundanta parenteser som är lätta att eliminera ska inte finnas:

- parenteser omkring hela uttrycket och omkring operander är lätt att undvika
- det är inte heller svårt att undvika parenteser kring högerledet i ett tilldelningsuttryck
- för övrigt får redundanta parenteser finnas

Följande exempel visar ett uttryck med full parentetrisering:

```
((x) = ((1) + ((2) * (4)))
```

get_infix() ska kunna åstadkomma följande, där parentesen kring 2 * 4 är redundant men får finnas:

```
x = 1 + (2 * 4)
```

Obs! Prioritetstabell eller extra information i trädnoderna för att stödja detta ska *inte* användas, även om det skulle möjliggöra eliminering av *alla* redundanta parenteser. *Använd enbart dynamisk typkontroll för att utifrån nodernas typ göra detta – det är en del av övningen!*

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

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

Tilldelningsoperatorn (=) sitter i rotnoden.

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.

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

Felsäker – undantagssäker – undantagsneutral

Att kod är felssäker innebär, i någon mening, att koden är robust mot fel som kan uppstå. Man brukar dela upp felsäkerhet i ”garantinivåer”, allt ifrån inga garantier alls, ”no guarantee”, till att inga fel kan inträffa, ”no-fail guarantee”. I grunden har det att göra med vad som är rimligt att åstadkomma vid fel – även om det vore möjligt att återställa allt kan det vara mycket kostsamt och man kan behöva nöja sig med mindre.

En klassisk problemkälla är dynamisk minnestilldelning, vilket görs av operatorerna **new** och **new**[]. **new** och **new**[], i sina vanliga former, kastar undantaget bad_alloc om de misslyckas. Programs robusthet mot undantag benämns *undantagssäkerhet*, ”exception safety” och man brukar nämna tre garantinivåer:

- ”no-throw guarantee” – inga undantag kastas
- ”basic guarantee” – inga resurser förloras (till exempel dynamiskt minne), inga objekt ”förstörs” (hamnar i något oklart tillstånd)
- ”strong guarantee” – ”basic guarantee" samt att antingen fullföljs en operation helt eller så bibehålls tillståndet som gällde innan operationen påbörjades

Undantagsneutralitet innebär att det undantag som kastas under en operation ska vidarebefordras till den som begärde operationen. Undantaget ska alltså inte absorberas eller bytas ut mot något annat slags undantag.

Klassen Variable_Table – endast för VG

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(ostream) 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 std::map för att implementera variabeltabellen. Se den givna filen Expression.cc, för exempel på hur map används.

Variable_Table går förstås utmärkt att testa separat.

Variabeltabellens inverkan på övrig kod

Beräkning av ett uttrycks värde görs av funktionen evaluate() i Expression, som i sin tur delegerar uppgiften till evaluate() i Expression_Tree, där den egentliga beräkningen görs. Konsekvensen av detta är att dessa funktioner måste känna till variabeltabellen, dels för att kunna ta reda på en variabels värde när den ingår i beräkningen av ett uttrycks värde, dels för att kunna lägga till en variabel i tabellen när den ”definieras” i ett tilldelningsuttryck. Expression och Expression_Tree behöver alltså ha tillgång till den variabeltabell som Calculator använder. Diskuteras ditt förslag till hur detta ska lösas med assistenten innan du kodar!

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.

Tips i samband med utformning av klasser och klasshierarkier

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 enskilt objekt eller 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.

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?
- Destruering – till exempel att säkerställa återlämning av dynamiskt minne i fall det förekommer.
- 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 successivt!*

Kontrollera mot de checklistor som finns!

Sammanfattning

Uttryckliga krav och anvisningar för betyg G och VG har angivits i samband med beskrivningarna av de olika klasserna, främst i form av vilka operationer som ska finnas och i många fall vad de ska heta. I något fall finns även uttryckliga krav på hur en operation ska implementeras. Se även den givna koden! *Följ noga det som anges!*

Under rubriken **Tips i samband med utformning av klasser och klasshierarkier** ovan, hittar du allmäna tips om viktiga saker att tänka på vid design av klasser och klasshierarkier. Utformningen av klasserna är mycket viktig! Det ingår som en del av uppgiften och kommer att ingå i betygsättningen.

Observera att det är fullt medvetet att klasserna som ska konstrueras inte är specificerade i detalj. Det ingår i uppgiften att utifrån specifikationerna och allmäna regler för konstruktion av klasser reflektera över klassernas detaljutformning och komma fram till bra lösningar.

Fel kan inträffa de klasser som du själv ska konstruera, även om mycket fångas redan i samband med kontrollen av infixuttrycken. Fel i egna klasser ska hanteras med undantag i analogi med hur det gjorts för de givna klasserna.

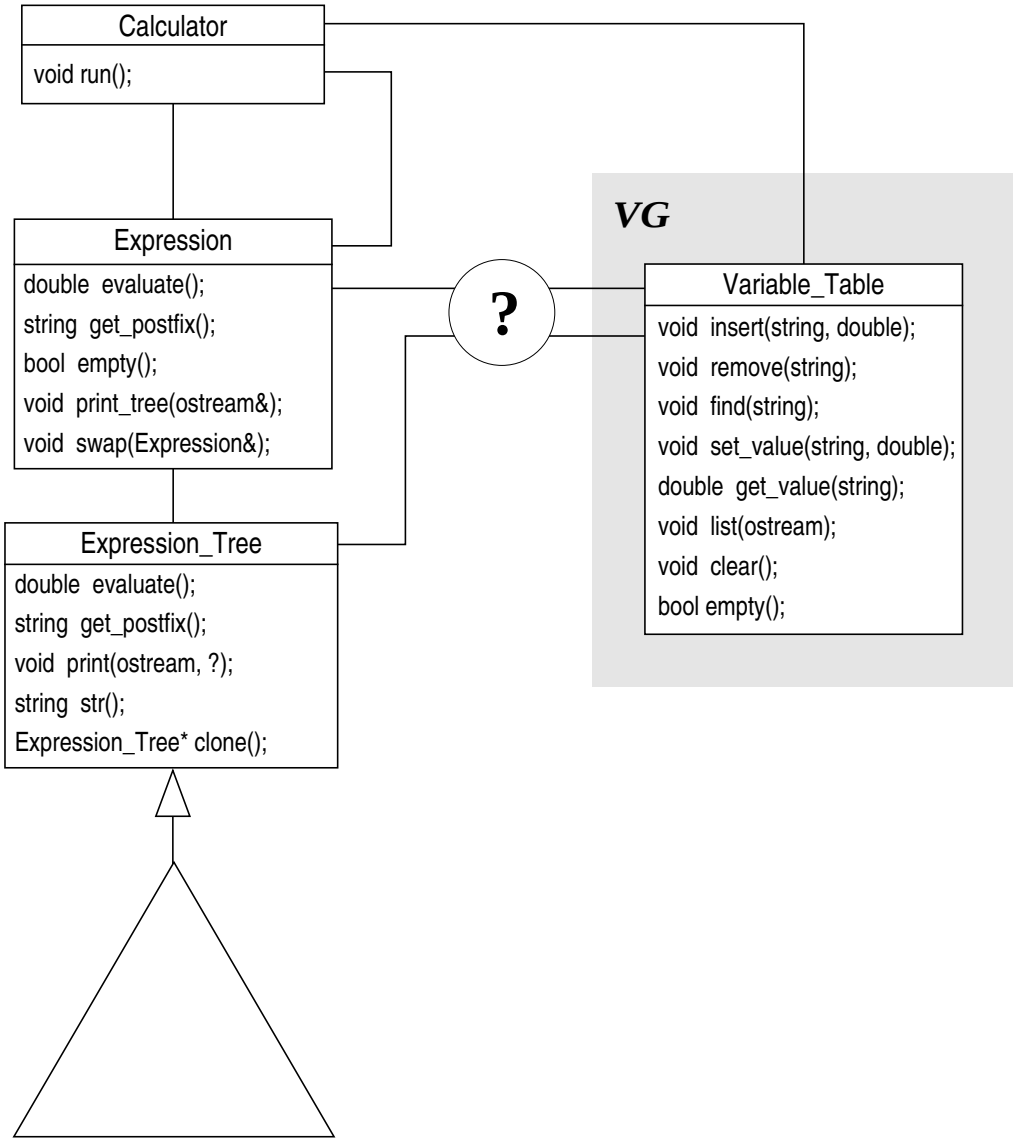
Klasserna Calculator, Expression och Expression_Tree med subklasser kommer att skilja sig lite åt i G- och VG-version. Klassen Variable_Table förekommer endast i VG-versionen av kalylatorn.

På nästa sida finns ett klassdiagram.

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.

Klassdiagram



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

Funktionaliteten i Expression och Expression_Tree motsvarar nivå G. Expression ska även ha en swap() som ej är medlem och dessutom tillhör make_expression() klassen Expression.

? – Calculator måste naturligtvis känna till Variable_Table men om och i så fall hur behöver Expression och Expression_Tree känna till Variable_Table