1 Introduktion

Det är meningen att man skall titta i detta häfte på datorn eftersom det innehåller ledtrådar som är gömda i filen (peka på dem så dyker det upp en liten ruta), det står

Ledtråd <1>

Kodlösningar kommer i vanlig ordning att läggas upp strax efter föreläsningen. Det finns många olika lösningar till övningarna, men det är rekommenderat att ni tittar på facit i efterhand. Ofta hittar man alternativa lösningar eller ett snyggare sätt att skriva.

2 Tips och ledning

Följande sidor visar lite mer detaljerade instruktioner för övningarna med tips. Tipsen poppar upp som ett litet fönster (annotation). Siffran inom parentesen för en ledtråd avser hur noggrann ledtråden är. Man bör titta på ledtråd 1 och 2 osv. i ordning. Försök lösa uppgiften med ledtråd 1 och gå över till 2 för mer information. Fastnar man, så fråga övningsledaren.

Eftersom kodningen i en konsol är på engelska (den kan inte läsa svenska tecken) håller vi oss till engelsk text. När det står ett utropstecken framför, betyder det oftast att det inte är en uppgift utan något att reflektera över.

3 Övning 2

I denna övning skall vi komplettera bråkuppgiften från övning 1. Skapa en ny fil, tänk på att filen skall sluta på .cpp och inte .c som **Code:Blocks** vill.

Meningen med uppgiften är att ni skall se hur man kan använda funktioner för att göra kod mer lättläst och snyggare.

Ni får skriva allting från början om ni vill, titta då gärna på första övningens facit för inspiration.

3.1 Bråk forts.

Eftersom vi inte gått igenom booleska variabler, så använder jag heltal, som ju tolkas som sanna då de är 1, falska om de är 0. I uppgift 1 (isInteger) och 4 (isZero) kan ni använda **bool** istället för heltal. En bool (kort för Boolean) har värdet **true** eller **false** och kan användas i if–satser eller **while** precis som man gör med **int**. Exempel på användning av bool:

```
bool flag = false;
```

det går bra att använda denna i en if–sats:

```
if(flag && a>0)
```

1. Skriv funktionen **isInteger** som tar en **double** och returnerar en **int**. Funktionen returnerar 0 om k inte är ett heltal, 1 annars.

Ledtråd <1> **Ledtråd <2>**

2. Skriv en funktion **cinIntegerBarrier** som läser in ett flyttal från konsol (via **cin**) och ger ett felmeddelande (via **cout**) till dess man skriver in ett heltal. Glöm inte att anropa funktionen från **main**.

Main är en funktion (den returnerar ett heltal). Programmet körs via **main** och allting som står där kommer att köras. En liknelse med **Matlab** vore att cpp–filen är som en m–fil och **main** är Matlab–prompten där man anropar andra funktioner.

Ledtråd <1>

3. Skriv funktionen **fractionExercise** som returnerar ett heltal.[1] Denna funktionen skall göra samma saker som som i Övning 1 uppgift 7, 9 och 10. Funktionen skall anropa **cinIntegerBarrier** för täljare och nämnare.

Ni får använda facit till övning 1 eller er egna kod för att göra denna uppgift. Meningen är att vi skall förkorta mängden kod från övning 1 och samtidigt förtydliga den logiska strukturen av programmet med hjälp av egenskrivna funktioner.

Anropa funktionen från main med en oändlig **while**. En oändlig loop kör så länge man inte stänger konsolfönstret eller trycker på **Ctrl-C**. Tänk på att använda **cinIntegerBarrier** och **isInteger** i er kod, den blir lättare att läsa då.

Exempel på kod:

```
cout << "Enter the numerator:";
cinIntegerBarrier();
cout << "Enter the denominator:";
cinIntegerBarrier();
```

4. Lägg till extra funktionalitet till funktionen **cinIntegerBarrier** så den även ger ett felmeddelande om man ger en nolla. Funktionen skall ta ett heltal som argument (isZero). Om isZero==1 så skall funktionen inte acceptera nollor annars är nollor ok.

Meningen är att anropet från **fractionExercise** skall skrivas:

```
cout << "Enter the numerator:";
cinIntegerBarrier(0);
cout << "Enter the denominator:";
cinIntegerBarrier(1);
```

På detta sätt bakar vi in mer funktionalitet i **cinIntegerbarrier** och gör därmed hela koden kortare.

Skriv även ut ett passande felmeddelande när nämnaren är noll och be användaren att skriva in ett nytt tal.

Ledtråd <1> **Ledtråd <5>**

6. Skriv funktionen gcd (eng. greatest common divisor) som räknar ut den största gemensamma delaren till två tal. Detta kan man göra med **while**. Man kan definiera gcd med *Euclides divisionsalgoritm* och nedanstående är en uträkning av

[1] Detta är mest för att avsluta funktionen på ett enkelt sätt.

gcd(24850, 7910)

$$\begin{aligned} 24850 &= 7910k_0 + \underbrace{1120}_{24850\ mod\ 7910} \\ 7910 &= 1120k_1 + \underbrace{70}_{7910\ mod\ 1120} \\ 1120 &= 70k_2 + 0 \end{aligned}$$

Upprepad användning av modulo ger slutligen en rest som är noll. Den sista icke försvinnande resten är vad vi är ute efter. För att förvissa sig om att funktionen är korrekt kan man anropa funktionen gcd i Matlab och jämföra. För att hantera negativa tal så kan ni utnyttja att

$$\gcd(a, b) = \gcd(|a|, |b|)$$

Ledtråd <1> **Ledtråd <2>** **Ledtråd <3>**

7. Övning 1 handlade om att testa att förkorta bråk med 2. I denna övning skall vi förbättra programmet och förkorta bråktal *så långt som möjligt*. Programmet skall verka smart och skriva ut det nya bråket om det finns en förkortning.

Om vi för samman ovanstående beräkningar för gcd(24850, 7910) får vi

$$\begin{aligned} 24850 &= \underbrace{7910}_{1120k_1+70} k_0 + 1120 \\ &= (70k_2k_1 + 70)k_0 + 70k_2 \\ &= k_0k_1k_2 70 + 70k_0 + 70k_2 \\ &= 70(k_0k_1k_2 + k_0 + k_2) \end{aligned}$$

Vilket visar att 70 delar båda talen, alltså gcd(24850, 7910) = 70.

Ledtråd <1> **Ledtråd <2>** **Ledtråd <3>**

! Vad blir svaret om vi förkortar 24850/7910 så långt som möjligt (se ovan), vad approximerar decimaltalet?

Extra: Om ni har tid över så kan ni göra så om indatat är negativa tal så får vi en smartare utskrift. Alltså

$$\frac{-a}{-b} \to \frac{a}{b}, \frac{a}{-b} \to \frac{-a}{b}.$$

Så om vi skriver in 2 och -12 så blir utskriften -1/6 och inte 1/-6.

3.2 Approximation av pi

Dessa uppgifter kan ni göra i mån av tid. Fokus är att öva på loopar och mer komplicerade uttryck. Öppna och modifiera koden i **Ex_2_3.cpp**.

! Vill man se fler decimaler i utskrifter så kan man skriva: **cout.precision(17);**.

1. Använd **while** för att approximera π med följande serie

$$\frac{\pi}{4} = \sum_{k=0}^{\infty} \frac{(-1)^k}{2k+1}.$$

2. Läs in antalet iterationer som programmet skall köra och skriv ut approximationerna.

Ledtråd <1>

3. Ändra koden så du **inte** använder powerfunktionen.

Ledtråd <1>

3.3 Extra information, approximationer

I nästa övning kommer vi approximera π på ett litet annorlunda sätt. Övningen kommer också involvera användning av Matlab och ett sätt att visualisera data genererat av ett C++program.

Man använder sig ofta av approximativa lösningar av system eftersom en exakt lösning kanske är svår att hitta eller inte finns alls. Iterativa lösningar (med while eller for) används exempelvis i fysikmotorer för spel.

Ett exempel på detta är kandidatarbetet "Bury The Needle" där jag skulle skriva en funktion som uppdaterade positionen för en kamera för varje bildruta. Jag använde en iterativ lösning till en ODE (Eulers metod) där en fjäder uppdaterade positionen för kameran via en for-loop.

I Projekt 1 kommer ni stöta på denna typ av *iterativ approximation*. Där kommer vi använda ekvationen för gravitation av två kroppar för att simulera en enkel –något tillrättalagd– galax. Vi får ut med denna simulering ett rutnät av positioner som vi sedan renderar med Rhino.

En del av denna och förra övningen har gått ut på att förkorta bråk. I dessa uppgifter är bråk exakt representerade med täljare och nämnare. Flyttalsstandarden (IEEE 754) (länk) som beskriver double och float kan inte definiera vissa bråk exakt. Talet 0.1, som vi hade lagrat som bråket 1/10, går alltså inte att representera exakt med flyttalsstandarden. Ett sätt att direkt se konsekvensen av flyttalsstandardens tillkortakommanden är att skriva följande

$$(0.1)^2 - 0.01.$$

Detta kan man göra i Matlab och svaret blir alltså inte 0, som man tror. Detta beror på att decimaltalet, som lagras binärt, har oändligt många decimaler (i basen 2). I vårt decimalsystem (basen 10) representerar vi ett tal, exempelvis 1.34 (1.34_{10} betyder "i basen 10") som

$$1 \cdot 10^0 + 3 \cdot 10^{-1} + 4 \cdot 10^{-2}$$

Skall vi skriva upp 1/3 i decimalform så tar aldrig decimalutvecklingen slut ($1/3 = 0.33333333\ldots$), detsamma gäller för den binära representationen av talet 0.1. Vi får att

$$0.1_{10} = 0.000110011001100\ldots_2.$$

Eftersom decimalutvecklingen är oändlig kommer alla försök att spara talet bli en approximation. För tal av dubbel precision lagras "endast" 15-16 decimaler (s.k. maskinnoggrannheten).

I exemplet ovan subtraherar vi två tal som är nästan lika, på grund av att felet uppkommer nära den 16:e decimalen så kommer felet i subtraktionen att bli stort (**länk**).

Att byta bas från 2 för att lagra tal hjälper inte, vi kommer endast flytta problemet till ett annat tal som inte går att representera exakt.

Varför använder man inte bråktal och exakt beräkning istället? Säg att vi skapar en ny typ som består av två heltal, täljare och nämnare. Det går ju, men vi behöver också representera **irrationella** tal som $\sqrt{2}, \pi, \ldots$ vilka aldrig kan beräknas exakt. Det finns dock bibliotek som sparar alla värden symboliskt, likt det som finns på symbolhanterande miniräknare[2]. Ett bibliotek med stöd för derivering och integrering bland annat, **SymbolicC++**. Se exempelkörningar med biblioteket (**länk**).

Trots detta används flyttalsstandarden i de flesta programmeringsspråken. Flyttalsberäkningar görs även på moderna datorer i hårdvara, chippet kallas **FPU** "floating point unit" som beräknar med flyttal snabbt. Begränsningen på antalet decimaler som flyttal sparar utgör sällan några praktiska beräkningsproblem. Värt att nämna är att symbolhanterande verktyg (exvis Maple) sällan används i praktiska tillämpningar. Dessa verktyg är långsamma i jämförelse (de har inte specialhårdvara), de använder troligtvis även minne flitigt, vilket gör att det går ännu långsammare.

[2] Miniräknare som Texas Instruments 89 (länk), eller datorpaket som Maple (länk).