Sex projektuppgifter

1. Löpande texten – en animation i konsolen

Skriv en C++ Console Application som simulerar en löpande text. Ta som exempel texten **`C++ är kul>`** som ska röra sig horisontellt från konsolfönstrets vänstra kant tills den ”träffar” på ett hinder, t.ex. ett kryss **`X`**. Texten ska börja från vänstra kanten. Krysset ska ligga nära den högra kanten (ca. 70-80 tecken borta). Dessa ögonblicksbilder ska illustrera animationen:

Ledning:

Skriv ut först krysset **`X`** i slutet av en tom rad (fylld med mellanslag). Anteckna hur många mellanslag ni har valt för att placera krysset från konsolens vänstra kant. Gå i samma rad tillbaka till radens början genom att använda escapesekvensen **`\r`** (carriage return). **`\r`** skickar tillbaka markören till början av samma rad, utan att byta rad (till skillnad från **`\n`**). Skriv sedan ut **`C++ är kul>`** som då blir textens initialposition – det som visas i den första ögonblicksbilden ovan. Om ni vill bekanta er mer med **`\r`**:s funktion gör experiment med det i ett annat program.

Rörelsen kan sedan simuleras t.ex. i en **`for`**-loop genom att i varje varv av loopen med ett antal **`\b`** ta bort texten som skrevs ut i förra varvet. Escapesekvensen **`\b`** (backspace) tar bort *ett* tecken till vänster om det aktuella tecknet, precis som tangenten backspace (←). Stega sedan med ett (eller flera) mellanslag, vilket kommer att bestämma rörelsens ”hastighet”. Skriv slutligen om texten **`C++ är kul>`**.

Beräkna antalet varv i **`for`**-loopen genom att ta hänsyn till textens längd och avståndet som kryss **`X`** har från vänstra kanten (som antecknats ovan). Har ni räknat rätt, kommer rörelsen att stoppas strax före krysset **`X`**, utan att ta bort det – liknande den tredje ögonblicksbilden ovan.

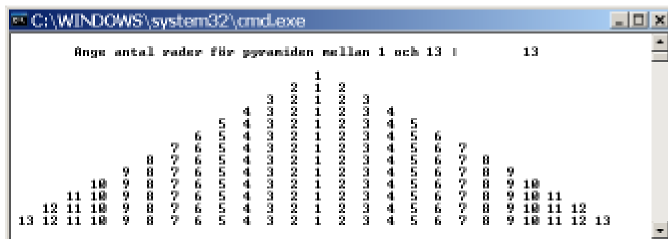
Även om ni gjort allt rätt kommer ni inte ”se” texten att röra sig, eftersom det går så fort, så att ögat inte hinner att se förloppet. Ni måste lägga in en fördröjning, vilket kan göras genom att infoga i loopen t.ex. satsen:

```
Sleep(100);
```

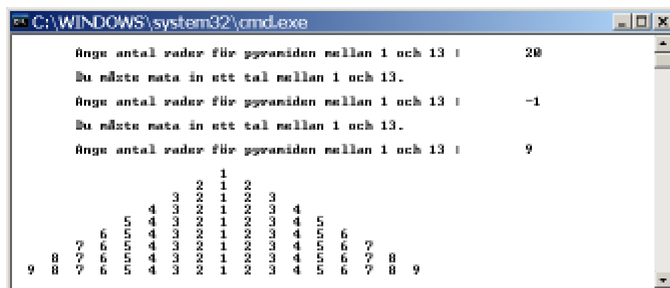
Parameterns enhet är millisekunder. Fördröjningsfunktionen **`Sleep()`** kräver inkluderingen av biblioteket **`windows.h`**.

2. Pyramiden

Slutmålet med detta uppdrag är att utveckla ett program som skriver ut en pyramidliknande figur med tal, som t.ex. ser ut så här:



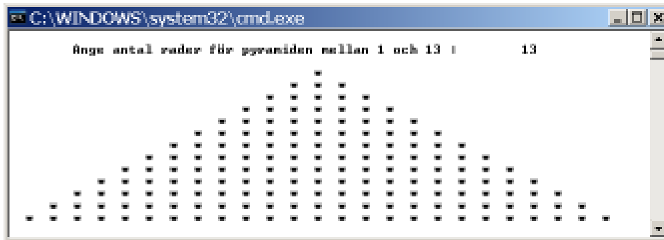
Programmet ska vara så generellt att det skriver ut talpyramider även om man matar in mindre antal rader. Uppmana användaren att hålla sig till talintervallet [1, 13]. Annars ryms talpyramiden inte i konsolen. Så här kan en körning se ut:



Ledning:

Denna ledning är endast en rekommendation och ska inte förhindra att ni använder egna idéer för att lösa problemet. Det finns andra möjliga tillvägagångssätt. Ni kan använda hela eller också delar av denna ledning för att komma igång.

Man kan *börja* med ett program som ritar en pyramid av *stjärnor* istället för tal:



Strunta till att börja med även på hanteringen av felinmatning av antal rader. Jobba med ett fast antal rader. Du kan lägga till det senare.

Använd en nästlad for-sats med en yttre loop och tre inre loopar:

- En för de tomma platserna i pyramiden (mellanslagen),
- En för stjärnorna i pyramidens högra halvan (räknat från den vertikala mittlinjen (symmetriaxeln),
- En för stjärnorna i pyramidens vänstra halvan.

Räkna med att ni måste använda i de inre looparna den yttre loopens räknare och slutvärde. T.ex. kan villkoret i den första inre loop som ritar de tomma platserna, se ut så här:

```
column <= numberOfRows - row;
```

Här är **`column`** den inre loopens, **`row`** den yttre loopens räknare och **`numberOfRows`** hela pyramidens antal rader, t.ex. 13. Då kan den första inre loopen skriva ut tre mellanslag i varje varv. I de två andra inre looparna kan två mellanslag och en **`*`** skrivas ut.

3. Kaffeautomaten

Du får i uppdrag att programmera en kaffeautomat. Uppdragsgivaren förväntar sig ett professionellt program som lätt kan uppdateras, om man skulle byta till en nyare automatmodell om något år. Därför anlitar man en objektorienterad programmerare. Skriv koden så generellt som möjligt så att programmet även kan modifieras för vilken varuautomat som helst, dessutom enkelt kan översättas till vilket programmeringsspråk som helst.

Programmet ska simulera en *aktion* i automaten, dvs det man *gör* med den. I händelsernas centrum ska finnas en klass som beskriver det som *pågår* i automaten, efter att användaren fått läsa menyn, valt en

dryck och stoppat in pengar. Deklarationen till en sådan klass kan – i stora drag – se ut så här:

```
class Coffee_action
{
   string productName;
   double price;
   double payment;
   double change;

   public:
      Coffee_action(char product, double money)
      {
         switch(product)
         {
            . . .
         }

         payment = money;
         change = payment - price;
      }

      void change_in_coins()
      {
            . . .
      }
};
```

Konstruktorn **`Coffee_action()`** ska initiera de privata datamedlemmarna **`productName`** och **`price`** beroende på valet av dryck och skriva ut ett meddelande om inlagt belopp samt drycken som ska levereras. Detta kan med fördel kodas med en **`switch`**-sats (ovan). Efter **`switch`**-satsen initieras även de privata datamedlemmarna **`payment`** och **`change`**.

Skriv ditt huvudprogram i en separat fil. Börja i **`main()`** med att skriva ut en meny över alla varor samt priserna, t.ex.:

K(affe)	12.00 kr
E(spresso)	14.00 kr
C(hoklad)	11.50 kr
L(Kaffe Latte)	13.00 kr
P(Cappuccino)	13.50 kr

Låt sedan användaren välja en dryck genom att läsa in begynnelsebokstaven till varorna ovan med en **`char`**-variabel. Låt användaren sedan lägga in pengar. Läs in beloppet till en **`double`**-variabel. Fortsätt med att skapa ett objekt av klassen **`Coffee_action`** inkl. anrop av konstruktorn. Vid detta anrop skickas till de inlästa värdna, dvs den valda varan samt det inlagda beloppet, som aktuella parametrar till konstruktorn **`Coffee_action()`**.

Ta hand om en ev. felaktig eller otillräcklig betalning från användarens sida genom att ge användaren möjlgheten att komplettera sin betalning.

Efter att objektet skapats och datamedlemmarna initierats via konstruktorn kan metoden **`change_in_coins()`** anropas som ska dela upp växeln i automatens ”tillåtna” myntslag (10-kr, 5-kr, 1-kr och 50-öringar) och skriva ut hur många av varje ”tillåtet” myntslag som ska ges tillbaka. För att åstadkomma detta kan följande algoritm användas:

Algoritm för omvandling av ett belopp till olika myntslag [1]

Eftersom denna algoritm endast fungerar för heltal, måste **`change`** som är ett belopp i kronor och ören av typ **`double`**, först räknas om till ett rent örebelopp av typ **`int`**, vilket kan göras genom att multiplicera det först med **`100`** och sedan omvandla till **`int`**:

```
int total = (int) (change * 100);
```

I fortsättningen kommer alltså den givna växeln att stå som ett *öre*belopp i **`int`**-variabeln **`total`**. Gör så här för att få antalen ”tillåtna” myntslag:

1. För att få antalet 10-kronor heltalsdivideras **`total`** med **`1000`** eftersom 10-kronor är **`1000`** ören:

```
int ten = total / 1000;
```

Hur många gånger ryms **`1000`** – eller 10-kronor – i **`total`**? Det antalet tilldelas till **`ten`**. Eller med andra ord: **`1000`** dras av från **`total`** så många gånger tills resten blivit mindre än **`total`**. Det antalet som tilldelas till **`ten`** blir antalet 10-kronor. Divisionen ovan är inte vanlig division utan heltalsdivision eftersom både **`total`** och **`1000`** är heltal. Dvs **`total`** divideras med **`1000`**, resultatet tas, resten ignoreras, t.ex. **`6975/1000`** ger **`6`**. Resten **`975`** ignoreras här, men används i fortsättningen.

2. För att få antalet 5-kronor divideras just *resten* som blev kvar från punkt **1** med **`500`** eftersom 5-kronor är **`500`** ören:

```
int five = (total % 1000) / 500;
```

Här används modulooperatorn **`%`**. ”Resten som blev kvar från punkt **1**” är just **`(total % 1000)`**. T.ex. **`6975 % 1000`** ger **`975`**. Efter att ha dragit av alla 10-kronor från **`total`** divideras resten med **`500`** för att få reda på hur många 5-kronor som finns i **`total`**. T.ex. **`975/500`** ger **`1`**. Resultatet av denna division ges till **`five`**, resten ignoreras och används i fortsättningen.

---

[*] Myntbetalningen inkl. behandlingen av 50-öringen beror inte på nostalgi utan på internationalisering. Vi vill hålla möjligheten öppen för en överföring av programmet till andra länder där automater med myntbetalning fortfarande finns. Även ett ev. byte till Euro eller andra valutor där den halva valutaenheten finns kvar, ska vara möjligt. Omvandlingen av växelbeloppet till automatens myntsystem inkluderar en programmeringsteknisk finess som kan vara värd att lära sig. Logiken inkl. användningen av modulooperatorn ligger till grund även för en generell omvandling av det decimala talsystemet till andra system.

I ytterligare tre steg kan de övriga formlerna för beräkning av antalet 1-kronor (**`one`**), 50-öringar (**`half`**) och resten i öre (**`rest`**) skrivas, när mönstret i algoritmen (förhoppningsvis) har trätt fram:

```
int one  =  ((total % 1000) % 500) / 100;
int half = (((total % 1000) % 500) % 100) / 50;
int rest = (((total % 1000) % 500) % 100) % 50;
```

Man tar förra stegets formel, ersätter **`/`** med **`%`** och lägger till en heltalsdivision med den nya enhetens örebelopp. I det allra sista steget däremot, där man är ute efter allra sista resten i öre, måste **`%`** användas hela vägen. Självklart är restörebeloppet inte av praktiskt intresse när automaten inte kan spotta ut det.

4. Frekvenstabell – stämmer sannolikhetsläran?

Följande program simulerar tärningskast genom att generera slumptal mellan **`1`** och **`6`**. Resultatet skrivs ut i tabellform.

```
// Dice.cpp
// Simulerar tärningskast: Slumpar fram tal mellan 1 och 6
// och skriver ut dem i en tabell. Nästlad for-sats
#include <iostream>
using namespace std;

int main()
{
   srand(time(0));              // Skapar variation i slumpen
   int r, k, rader, kolumner;
   cout << "\nAnge antal rader och kolumner (t.ex. 10 15): ";
   cin >> rader >> kolumner;
  cout<<"\nDet blir "<< rader*kolumner<<" tärningskast:\n\n";
   for (r=1; r<=rader; r++)     // Låter en rad att skrivas ut
   {                            // och byter rad.
     for (k=1; k<=kolumner; k++)// Skriver ut den r:te raden.
         cout << 1 + rand() % 6 << "   ";
     cout << '\n';
   }
}
```

Testa programmet **`Dice`** och vidareutveckla det. Det nya programmet ska undersöka sannolikhetsläran sats om att i idealfallet sannolikheten för ett utfall vid tärningskast är **1/6** och att det praktiska resultatet närmar sig idealfallet, ju större antalet slumpförsök blir. Genomför denna undersökning genom att ställa upp en *frekvenstabell*. Nedan beskrivs frekvenstabellen:

Frekvens är *antalet* förekomster av ett resultat (utfall) bland tärningens **`6`** möjliga.

Låt programmet genomföra olika antal simuleringar och räkna vid varje simulering frekvensen för varje resultat **`1`**, ... ,**`6`** av tärningskastet. T.ex. ska man kunna läsa av från tabellen hur många gånger resultatet **`1`** förekommer när man kastar tärningen

50 gånger, 100 gånger, 1 000 gånger, 5 000 gånger, 10 000 gånger, osv. Avgör själv hur långt du går. Samma information ska man kunna läsa av från tabellen om tärningskastets andra resultat `2`, ... ,`6`.

Infoga i tabellen även en kolumn som för varje resultat av tärningskastet visar kvoten:

Frekvens / Antalet tärningskast

Denna kvot är den experimentella sannolikheten för ett visst resultat.

Undersök på vilket sätt den experimentella sannolikheten närmar sig den ideala sannolikheten för varje resultat, som enligt sannolikhetsläran borde vara **1/6** eller **0,16667**.

5. Palindrom – en lek med ord

En *palindrom* är en sträng som inte ändras när den läses baklänges. T.ex. är orden *rar*, *död* och *radar* palindromer, även namnet *Hannah* när det stavas så. Men även en text som *ni talar bra latin* är en palindrom om man ignorerar mellanslagen. Och det ska man göra. Därför: vid behandling av sådana texter i ett program låt koden först ta bort alla mellanslag.

Skriv en funktion **`bool palindrom(char *a)`** som avgör om en sträng är en *palindrom* eller ej. Anropa sedan funktionen i **`main()`** efter inmatning av en sträng som ska testas, i ett C++ program som hanterar strängar med pekare. Låt användaren mata in strängar – med eller utan mellanslag – så länge tills man hittat en palindrom. Bjud på möjligheten att avsluta om ingen palindrom hittas och föreslå användaren ett antal palindromer.

6. Collatz algoritmen – rekursiv

I kursen *Programmering med C++* behandlades *Collatz algoritmen* som alltid slutar med 1 oavsett startvärde – ett empiriskt resultat som matematiskt är obevisat:

Tänk dig ett positivt heltal (startvärde). Är talet udda multiplicera det med 3 och addera 1. Är talet jämnt dividera det med 2. Gör samma sak med resultatet. Fortsätt **tills** du fått 1.

Då implementerade vi Collatz algoritmen *iterativt* med en **`do`**-loop:

```
#include <iostream>
using namespace std;

int main()
{
   int no;
   cout << "\n\tMata in ett pos.heltal:\t";
   cin >> no;
   cout << "\n\t" << no;

   do
   {
      if (no % 2 == 1)
         no = 3 * no + 1;
      else
         no = no / 2;
         cout << "\n\t" << no;
   } while (no != 1);

   cout << "\n";
}
```

Skriv ett C++ program som implementerar Collatz algoritmen med en rekursiv funktion och anropar den från main().

Ledning: Läs kap ***2.6 Rekursion*** i kursboken, sid 45-48.

Undersök om fenomenet *beräkningskomplexitet* (sid 48) som observerades i Fibonacci rekursionen, även uppträder här. Förklara orsaken. Resonera om för- och nackdelarna av Collatz algoritmens iterativa och rekursiva implementering.