Inlämningsuppgift 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 [‡]

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;
```

---

[*] 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.

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.

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 restöre-beloppet inte av praktiskt intresse när automaten inte kan spotta ut det.