Inlämningsuppgift 1

Gymnastiktävling Skriv ett C++ program som avgör en tävling i gymnastik. Tre tävlande deltar i tävlingen. De får sina poäng av tre olika domare. Poängen ska ligga mellan **1** och **100**. Varje tävlandes poäng ska summeras till en totalpoäng. Sedan ska programmet bestämma den största totalpoängen samt ska skriva ut både varje tävlandes totalpoäng, den största totalpoängen och tävlingens vinnare.

Använd tre arrays. Varje array ska lagra poängen för varje tävlande. Varje element i arrayen ska tilldelas en domares poäng. Simulera domarnas poänggivning med slumptal inom intervallet **[1, 100]**. Bestäm den största totalpoängen bland de tre tävlandes totalpoäng.

Frivilligt: I fall att två tävlande får samma antal totalpoäng, utropa båda som tävlingens vinnare.

Ledning:

Steg 1 Läs i kursboken, avsn. *4.7 Arrays* (sid 89).

Steg 2 Läs om slumptal i kursboken, avsn. *4.9 Hantering av slumptal* (sid 95), speciellt om *Slumptal inom ett intervall* (sid 96).

Steg 3 Skapa tre arrays, en för varje tävlandes poäng. Varje array ska innehålla 3 element.

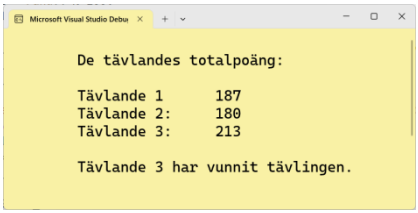
Steg 4 Fyll varje array med slumpvärden mellan 1 och 100 motvarande de tre domarnas poäng.

Steg 5 Skapa tre variabler för totalpoängen, en för varje tävlande, och tilldela dem summan av varje tävlandes poäng.

Steg 6 Bestäm den största bland de tre tävlandes totalpoäng. Använd funktionen **`max()`** som behandlas i kursboken (sid 102).

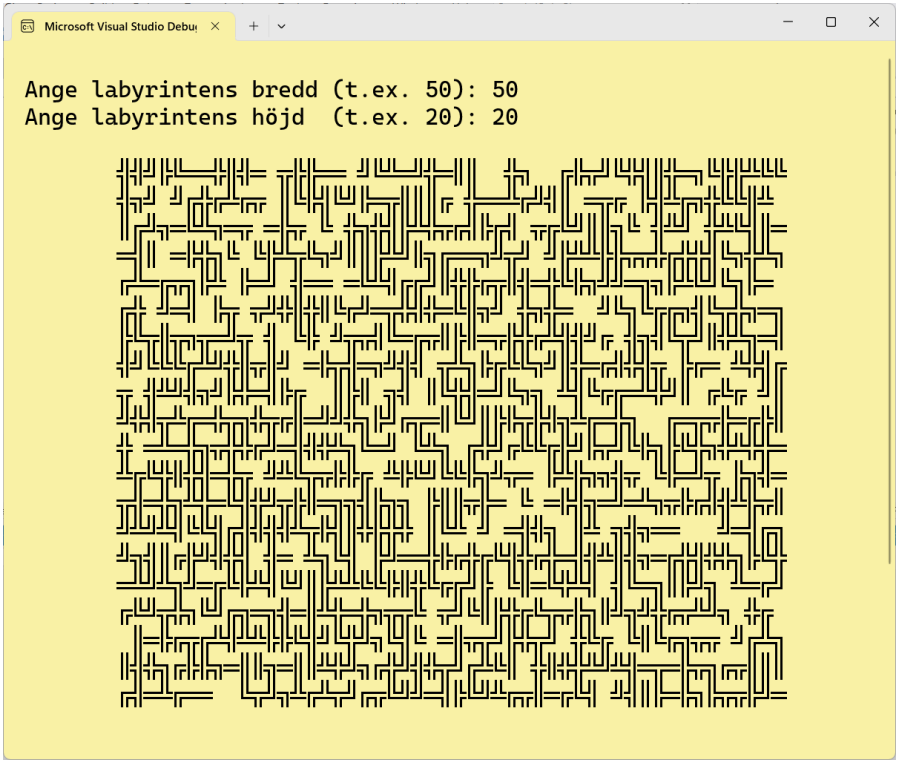
OBS! För att bestämma tävlingens vinnare, borde du lägga till lite kod till funktionen **`max()`**, t.ex. i form av en **`string`**-variabel **`winner`**, så att den inte bara hittar den största totalpoängen (**`max`**) utan även den tävlande (**`winner`**) som fått denna totalpoäng.

Steg 7 Skriv ut varje tävlandes totalpoäng och gymnastiktävlingens vinnare. Ex.:



Inlämningsuppgift 2

Labyrinten Visst är det roligt att med ett C++ program låta datorn rita en labyrintartad figur på skärmen som kan se ut så här:



Visserligen är detta ingen riktig labyrint. För en sådan skulle det krävas mycket mer. En riktig labyrint skulle kunna vara t.ex. grafikdelen till ett spelprojekt med lite andra finesser, färger osv. En sådan avancerad figur kan inte ritas i konsolen.

Bilden ovan är ”ritad” i *text mode* och visar snarare om en *labyrintartad figur* som är slumpmässigt ihopsatt av ett antal tecken som vi kallar för *dubbla linjegrafiktecken (LGT)* samt mellanslaget. I figuren ovan är de slumpade i en 2D utskrift, en slags tabell eller matris med 50 rader och 20 kolumner. I koden åstadkommer man detta med en nästlad **`for`**-loop, där den yttre loopen skriver ut raderna och den inre loopen kolumnerna, se avsn. ***6.8 Nästlade for-satser***, sid 156. Tecknen i figuren ovan är slumpvist valda bland de dubbla LGT och mellanslaget. Därför borde varje körning av programmet generera en lite annorlunda labyrintartad figur.

Du kan gärna försöka med en egen algoritm att skriva ett C++ program som ritar en sådan figur. Men i instruktionerna som följer har du i alla fall ett förslag till en algoritm (***Steg 1-5***) som fungerar.

Algoritmen

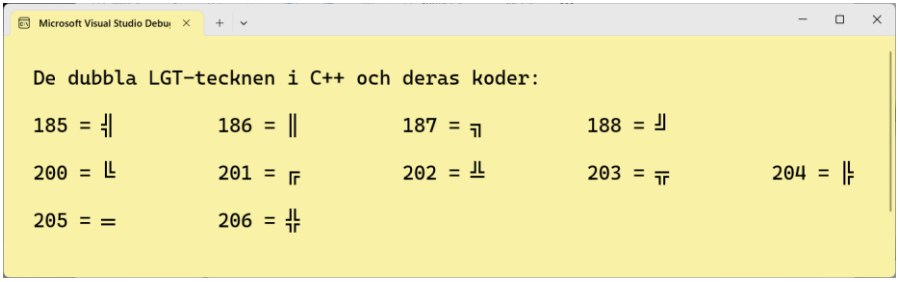
Steg 1 Bekanta dig med hantering av tecken i C++ inkl. explicit typkonvertering (sid 120), genom att experimentera med följande program:

```
// Int2char.cpp
// Ger tecknet till en inmatad kod
// Representation av tecken med heltalskoder
#include <iostream>
using namespace std;

int main()
{
   int code;
   cout << "\nMata in ett heltal: ";
   cin >> code;
   cout << "\nDet inmatade heltalet är " << code <<
        " och är koden till tecknet " << (char) code << "\n";
}
```

Kör programmet ovan genom att mata in koderna **185-188** och **200-206**.

Steg 2 För få en översikt över alla dubbla LGT samt deras koder i C++ skriv ett program som producerar följande utskrift:



Dessa tecken finns i den utvidgade delen av teckentabellen och används för att rita raka linjer, ramar, tabeller, skisser osv i en textbaserad miljö *(text mode)*. De används tillsammans med mellanslaget för att rita labyrinten. Jämför koderna med utskriften till programmet **AsciiFor** (sid 154).

Steg 3 Bekanta dig med hantering av slumptal i avsn. ***4.9 Hantering av slumptal*** (sid 95), speciellt om ***Slumptal inom ett intervall*** (sid 96).

Steg 4 Skriv ett program som med hjälp av C++:s slumpgenerator och en nästlad **for**-sats ritar labyrinten, en slumpmässigt ihopsatt figur bestående av de dubbla LGT samt mellanslaget.

Steg 5

Deklarera en teckenvariabel **`letter`** och en heltalsvariabel **`randNo`**. Initiera **`letter`** till mellanslaget, dvs **`' '`**. Låt **`randNo`** anta slumpvärden mellan **`0`** och **`11`** med satsen:

```
randNo = rand() % 12;
```

Fortsätt med följande **`if`**-satser:

Om **`randNo`** blir **`0`**	ska **`letter`** tilldelas	1:a LGTs teckenkod.
Om **`randNo`** blir **`1`**	ska **`letter`** tilldelas	2:a LGTs teckenkod.
Om **`randNo`** blir **`2`**	ska **`letter`** tilldelas	3:e LGTs teckenkod.
.	.	
.	.	
.	.	
Om **`randNo`** blir **`9`**	ska **`letter`** tilldelas	10:e LGTs teckenkod.
Om **`randNo`** blir **`10`**	ska **`letter`** tilldelas	11:e LGTs teckenkod.
Om **`randNo`** blir **`11`**	ska **`letter`** tilldelas	mellanslaget, dvs **`' '`**.

Numreringen av LGT-teckenkoderna avser den ordning som är föregiven i teckentabellen, se utskriften på förra sidan. I övrigt fungerar vilken numrering som helst.

Observera att LGT-teckenkoderna är av typ **`int`**, medan variabeln **`letter`** är av typ **`char`**. Vid tilldelningen konverteras **`int`** automatiskt till **`char`**. Vid utskriften med **`cout`** skrivs ut tecknet, inte koden. Mellanslaget (**`' '`**) däremot är från början av typ **`char`**, så att vi inte behöver bry oss om koden. Se upp att **`''`** inte är mellanslaget utan den tomma strängen och ger kompileringsfel.

Skriv ut **`letter`**, så att du får ETT tecken, antingen ett LGT eller mellanslaget. Testa även om du vid varje körning får *olika* LGT eller mellanslaget.

Först när allt detta fungerar låt de 12 **`if`**-satserna ovan samt tilldelningen av variabeln **`randNo`** ingå i en enkel *loop*, t.ex. en **`for`**-sats, för att rita labyrinten.

Ersätt den enkla loopen med en nästlad **`for`**-sats och lägg in ett radbyte mellan den inre och yttre slingan för att kunna styra labyrintens storlek (höjd och bredd) vid varje körning. Låt användaren ange höjd och bredd.

Extrauppgift (frivilligt):

Ersätt de 12 **`if`**-satserna ovan med en annan konstruktion: Samla alla dina LGT-koder och mellanslaget i en *array* och låt slumpen ta ut tecken ur denna array. För utskriften kan du fortsätta med att använda nästlad **`for`**-sats. Det blir en kortare och elegantare kod.

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 inte simulera själva automaten utan en *aktion* i automaten, dvs snarare det man *gör* med den. I händelsernas centrum ska finnas en *klass* som beskriver det som pågår i automaten *efter* att användaren stopppat in pengar i den och valt en dryck. Deklarationen till denna klass kan – i stora drag – se ut så här:

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

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

         payment = money;
         change = payment - price;
      }

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

Konstruktorn **`Coffee_action()`** ska tilldela de by default privata datamedlemmarna **`productName`** och **`price`** värden beroende på valet av dryck och skriva ut ett meddelande om inlagt belopp samt drycken som ska levereras. Detta kan

kodas med hjälp av en **switch**-sats (ovan). Men istället för **switch** kan man lika bra använda nästlade **if-else**-satser. Skapa objekt av klassen **Coffee_action** i en annan klass i en separat fil som endast innehåller **main()**.

Börja i **main()** med att skriva ut en meny över alla varor samt priserna, t.ex.:

K(affe)	8.00 kr
E(spresso)	9.50 kr
C(hoklad)	7.50 kr
L(Kaffe Latte)	9.00 kr
P(Cappuccino)	9.50 kr

Låt sedan användaren lägga in pengar. Läs in beloppet till en **double**-variabel. Låt användaren även välja en dryck genom att läsa in begynnelsebokstaven till varorna ovan med en **char**-variabel. Sedan kan ett objekt av den ovan deklarerade klassen **Coffee_action** skapas in inkl. anrop av konstruktorn **Coffee_action()**. Vid detta anrop skickas de inlästa värdena till det inlagda beloppet och den valda varan som aktuella parametrar till **Coffee_action()**. Efter att objektet skapats och datamedlemmarna initierats kan metoden **change_in_coins()** anropas.

Komplettera programmet med att ta hand om en eventuellt felaktig eller otillräcklig betalning från användarens sida. **change_in_coins()**[*] är en metod 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. Växelbeloppet måste omvandlas till detta mynt”system”. För att åstadkomma det, 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 avrunda resultatet till heltal:

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

I fortsättningen kommer alltså den givna växeln att stå som ett *öre*belopp i **int**-variabeln **total**. Anledningen till konverteringen till **int** i satsen ovan är att den fördefinierade metoden **Round()** som avrundar till närmaste heltal, ändå returnerar ett värde av typ **double**.

---

[*] 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.

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.

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.