Slutuppgifter

Slutuppgiftsperiod på våren: Under en längre period på våren kommer vi att fokusera enbart på slutuppgifterna. Denna tid är avsatt för att du ska kunna arbeta koncentrerat och strukturerat med din slutuppgift. Du får möjlighet att planera, koda och förbättra ditt program. Det är viktigt att du utnyttjar denna period för att visa kontinuerliga framsteg och följa den plan du lagt upp.

Viktiga punkter att tänka på:

1. **Arbetsprocess och kontinuitet**: Först och främst, kom ihåg att detta **inte** är en inlämningsuppgift där du bara lämnar in ett färdigt resultat. Det är viktigt att jag kan följa din arbetsprocess **under lektionstid**. Om du bara lämnar in uppgiften vid slutet av kursen utan att visa kontinuerligt arbete, kommer uppgiften **inte att bedömas**. Endast kontinuerligt arbete räknas, inte en inlämning i sista stund.
2. **Inte bara kod**: Det räcker inte att bara göra uppgiften. Du måste också vara beredd att:
    - Förklara **hur** du har tänkt under arbetet.
    - Svara på frågor om **genomförandet**.
    - Diskutera hur programmet kan **utvecklas vidare**.
3. **Bedömning av svårighetsgrad**: Varje uppgift har en bedömning av svårighetsgraden. Välj uppgifter som motsvarar ditt betygsmål. **Hårt arbete** med flera uppgifter på medelnivå kan ge högre omdöme än att bara göra en svår uppgift halvdant.
4. **Komplexitet och tidsåtgång**: Vissa uppgifter går inte att färdigställa helt under den tid du har. Det är okej. Ju längre du kommer och ju närmare du är en färdig lösning, desto bättre blir ditt resultat.
5. **Planering är nyckeln**: Innan du börjar programmera, **ska du göra en planering** över hur du tänker lösa problemet. Det är viktigt att du inte försöker lösa alla problem på en gång, speciellt om uppgiften är mer komplicerad. Bryt ned problemen i mindre delar.
6. **Välj rätt svårighetsgrad**: Självinsikt är viktigt. Att börja med en uppgift på avancerad nivå, för att sikta på ett högre betyg, är inte en bra strategi om du inte känner att du har rätt kunskapsnivå. Det är bättre att börja med en enklare uppgift, göra klart den, och sedan gå vidare till svårare uppgifter.
7. **Egen kodning**:
    - Du **får inte ladda ner färdigskriven kod** eller använda kod som någon annan har skrivit åt dig.

- Jag kommer att **ställa frågor** om din kod och eventuellt be dig göra ändringar medan jag sitter bredvid. Om du inte har skrivit koden själv kommer det att bli tydligt vid den punkten.
- Det är viktigt att du **själv** skriver all kod. Om du tycker att uppgiften är för svår, börja med en enklare uppgift. Om även de enklare uppgifterna är svåra, bör du gå tillbaka och öva på tidigare kursmoment för att stärka dina grundkunskaper.

8. **Snygghet är inte nödvändigt, men kan vara givande:** Det är inte ett krav att skapa ett grafiskt användargränssnitt (GUI) för ditt program, men det kan vara ett roligt sätt att göra programmet mer tilltalande och användarvänligt. Grundregeln är att du först ska lösa uppgiften i ett textbaserat gränssnitt.
9. **Kommentera din kod (ett krav)**: Kommentering av koden kommer att vara en del av bedömningen. Särskilt när era program blir längre, är det viktigt att vänja sig vid att kommentera koden regelbundet. Kommentarer hjälper både dig själv och andra att förstå hur ditt program fungerar, särskilt vid mer komplicerade delar av koden. **Viktiga delar av ditt program** ska ha tydliga kommentarer som beskriver vad de gör. Detta underlättar både din egen genomgång och bedömningen, och gör det lättare att följa din tankeprocess.
10. **README-fil:** Till en av dina slutuppgifter, ska du redovisa en README-fil på svenska eller engelska som kortfattat beskriver projektet och ger en överblick över dess funktion och användning. Följ strukturen nedan när du skriver din README.

    1. **Sammanfattning (Summary)**

       Namnge ditt projekt och beskriv det kortfattat i 2-3 meningar. Vad gör ditt program? Håll beskrivningen enkel och lättförståelig.

    2. **Bakgrund (Background)**

       Vilket problem kommer din idé att lösa? Hur vanligt eller frekvent är detta problem? Vad är din personliga motivation för att välja detta projekt? Varför tycker du att detta ämne eller problem är viktigt eller intressant att lösa med programmering?

    3. **Nyckelaspekter (Key aspects) (frivilligt)**

       Om du vill, kan du beskriva några viktiga funktioner eller tekniska detaljer som du tycker är särskilt värda att lyfta fram. Här kan du till exempel nämna något intressant om hur programmet är uppbyggt eller hur du löste ett specifikt problem.

    4. **Hur används programmet (How is it used?)**

       Tänk på detta som en enkel användarmanual för ditt program. Beskriv hur man kör ditt program och hur det fungerar. Om det är relevant, kan du ge exempel på vad som händer när programmet

körs och vad användaren kan förvänta sig för resultat. Om det finns speciella kommandon eller inmatningar användaren måste göra, förklara dessa här.

5. **Utmaningar (Challenges)**

   Vilka delar av problemet löser ditt projekt inte? Det är viktigt att förstå att alla tekniska lösningar har sina begränsningar.

6. **Vad härnäst (What next?)**

   Hur kan ditt projekt utvecklas vidare? Vilka funktioner skulle du kunna lägga till för att förbättra programmet? Finns det möjligheter att göra det mer användarvänligt, effektivt eller spännande? Reflektera över hur projektet kan växa och bli något ännu bättre.

En lämplig textlängd för **README-filer** är att de inte överstiger **1-2 sidor** i text, eller cirka **300-500 ord**.

Exempel på README för ett Sten-sax-påse-spel:

1. Sammanfattning
Sten-sax-påse är ett textbaserat spel där spelaren möter datorn i ett klassiskt matchande-spel. Spelaren väljer mellan sten, sax eller påse, och datorn gör ett slumpmässigt val. Programmet jämför valen och avgör vem som vinner.

2. Bakgrund
Sten-sax-påse är ett enkelt och roligt spel som alla känner till. Genom att skapa detta spel kan jag träna på att använda villkorsstyrd logik och slumpmässiga val i programmering. Det är ett bra exempel på hur man kan arbeta med inmatning från användaren och jämföra olika val för att avgöra ett resultat.

3. Nyckelaspekter *(frivilligt)*
En viktig aspekt av programmet är hur datorns val genereras slumpmässigt med hjälp av en slumptalsgenerator.

4. Hur används programmet?
För att spela spelet, starta programmet och välj mellan sten, sax eller påse genom att skriva in ditt val när du blir ombedd. Datorn gör samtidigt sitt val, och programmet visar resultatet av omgången. Spelet kan spelas flera gånger tills användaren väljer att avsluta.

Exempel på körning:

```
Välj: sten, sax eller påse: sten

Datorn valde: sax

Du vann! Sten slår sax.

Vill du spela igen? (j/n): j
```

5. Utmaningar
Programmet hanterar endast ett enstaka spel åt gången. Det saknar avancerade funktioner som poängräkning eller möjlighet att spela mot en annan människa. Dessutom finns ingen felhantering om användaren skriver in ogiltiga värden.

6. Vad härnäst (What next?)
I framtiden kan spelet utvecklas genom att hålla räkningen på hur många gånger varje spelare har vunnit. En annan förbättring kan vara att implementera en meny där spelaren kan välja mellan olika spellägen, till exempel att spela mot en mänsklig motståndare.

Inlämning och redovisning:

- **Muntlig redovisning:** Du kommer att redovisa din lösning muntligt och förklara hur du har arbetat med uppgiften. Du bör vara beredd på att svara på frågor om koden.
- **Inlämning av kod:**
    - **Alternativ 1:** Lämna in din kod som ett kodblock i ett Google-dokument. Använd formatet ***Byggsten-kodblock (C++)*** för att tydligt visa din kod.
    - **Alternativ 2:** Länka till ditt program om du har en **GitHub-repository,** så att koden kan granskas direkt i din repository.

Välj en eller flera av nedanstående uppgifter.

1.Omvandla sekunder till år, dagar, timmar, minuter och sekunder.

***Svårighetsgrad: Enkel nivå*.**

Skriv ett C++-program som tar emot ett heltal från användaren som representerar ett antal sekunder. Programmet ska sedan omvandla dessa sekunder till motsvarande antal år, dagar, timmar, minuter och sekunder.

- Programmet ska fungera för alla värden ≥ 0.

Exempel på inmatning och utmatning:
Inmatning: 987654321 sekunder
Utmatning:
Det motsvarar: 31 år, 251 dagar, 7 timmar, 15 minuter, 21 sekunder.

2. Generera Fibonaccis serie med hjälp av en array.

***Svårighetsgrad: Enkel nivå*.**

Ditt uppdrag är att skriva ett C++-program som beräknar och skriver ut de första N talen i Fibonaccis serie. Fibonaccis serie börjar med 0 och 1, och varje efterföljande tal är summan av de två föregående talen. Till exempel är de första 6 talen i serien: 0, 1, 1, 2, 3, 5.

Programmet ska:

1. Be användaren om att mata in hur många tal av Fibonaccis serie som ska beräknas.
2. Lagra talen i en array.
3. Skriva ut talen på en rad, separerade med mellanslag.
4. Hantera felaktig inmatning genom att ge ett lämpligt felmeddelande om användaren anger ett tal för N som inte är positivt eller som överskrider den maximala gränsen.

Exempel på körning och resultat:

```
Hur många tal i Fibonaccis serie vill du generera? 6
Fibonaccis serie: 0 1 1 2 3 5
```

3. Decimalt till binärt.

***Svårighetsgrad: Enkel nivå*.**

Skriv ett C++-program som omvandlar ett heltal i det decimala talsystemet till dess motsvarighet i det binära talsystemet.

- Programmet ska läsa in ett positivt heltal från användaren.
- Programmet ska omvandla det decimala talet till binärt och sedan skriva ut det binära talet.
- Om användaren matar in ett negativt tal ska programmet skriva ut ett felmeddelande och be om ett nytt tal.

Exempel på inmatning och utmatning:

Inmatning: 19
Utmatning: Det binära talet är: 10011

4.Decimalt till hexadecimalt.

***Svårighetsgrad: Enkel nivå*.**

Skriv ett C++-program som omvandlar ett positivt heltal i det decimala talsystemet till dess motsvarighet i det hexadecimala talsystemet.

Exempel på inmatning och utmatning:

Inmatning: 255
Utmatning: Det hexadecimala talet är: FF

Tips:

- För att omvandla ett decimaltal till hexadecimalt använder du modulusoperatorn (%) för att ta reda på resten när du delar talet med 16. Resterna representerar de hexadecimala siffrorna (0-9 och A-F).
- Du kan skapa en array eller sträng där varje index motsvarar en hexadecimalsiffra, där 0 motsvarar 0, 10 motsvarar A, och så vidare upp till 15 som motsvarar F.
- Efter att ha beräknat resten, dela talet med 16 och upprepa tills talet är 0.

5.Valutakursomvandlare.

***Svårighetsgrad: Enkel nivå*.**

Skriv ett program som omvandlar ett belopp från svenska kronor (SEK) till en annan valuta, baserat på en valutakurs vald från en meny. Valutakurserna ska vara sparade i en fil som programmet läser in vid start.

Valutakurser: Valutakurserna ska vara sparade i en fil, exempelvis "valutakurser.txt", med följande format:

```
USD 0.11
EUR 0.095
GBP 0.085
NOK 1.05
```

Exempel på inmatning och utmatning:

```
Välj valuta för omvandling från SEK:
1. USD (Amerikanska dollar)
2. EUR (Euro)
3. GBP (Brittiska pund)
4. NOK (Norska kronor)
Ange ditt val (1-4): 2
Ange belopp i SEK: 100
100 SEK motsvarar 9.5 EUR.
```

6.Korsord.

Svårighetsgrad: Enkel nivå/Medelnivå.

```
Ledtrådar:
Rad 1: Dialektal form av ordet sommar.
Rad 2: Svensk popartist som slog igenom på 1980-talet.
Rad 3: Namnet på en by och kommun i Normandie, Frankrike.
Rad 4: Svenskt tv-program känt för sin undersökande journalistik.
Rad 5: Rum och kök.
Kolumn 1: Ett uråldrigt verktyg som använder skuggor för att visa tid.
Kolumn 2: (0, 0).
Kolumn 3: Används för att styra markören på en dator.
Kolumn 4: I sportsammanhang en förbjuden dopningssubstans.
Kolumn 5: Norrköpings tidningar.
I I I I I I I I
I S O M E R I
I O _ _ _ _ I
I _ _ _ _ _ I
I _ _ _ _ T I
I _ _ K _ _ I
I I I I I I I I
Ange rad (2-5):
```

Specifikationer:

- **Spelplanen** är ett rutnät med minst 5x5 rutor.
- Användaren ska kunna **gissa en bokstav i taget** genom att ange rad, kolumn och bokstav.
- Spelet ska ge **ledtrådar** för både horisontella och vertikala ord.

- Spelet avslutas när alla ord är korrekt ifyllda.

Hjälpmedel:

- **Ledtrådar**

  Se ovan.

- **Spelplan**

	1	2	3	4	5
1	S	O	M	E	R
2	_	_	_	_	_
3	_	_	_	_	_
4	_	_	_	_	T
5	_	_	K	_	_

- **Facit**

	1	2	3	4	5
1	S	O	M	E	R
2	O	R	U	P	-
3	L	I	S	O	N
4	U	G	-	-	T
5	R	O	K	-	-

Förslag på utökning:

***Svårighetsgrad: Medelnivå*.**

- Gör så att programmet läser in korsord (ledtrådar, spelplan och facit) från filer, så att du kan byta korsord utan att ändra koden, annat än att läsa in en annan fil (eller andra filer).

7.Lotto-program.

***Svårighetsgrad: Enkel nivå*.**

Skriv ett program där användaren får skapa en lottorad genom att välja 7 unika nummer mellan 1 och 35. Programmet ska slumpa fram en rätt lottorad och sedan jämföra användarens rad med den rätta raden för att räkna antal rätt. Baserat på antal rätt ska programmet visa om användaren har vunnit och i så fall hur mycket.

Specifikationer:

- **Antal nummer i varje rad:** 7
- **Nummerintervall:** 1 till 35

- **Vinster:**
    - **7 rätt:** 10 000 000 SEK
    - **6 rätt:** 500 000 SEK
    - **5 rätt:** 10 000 SEK
    - **4 rätt:** 100 SEK
    - **3 rätt:** 50 SEK
    - **Mindre än 3 rätt:** Ingen vinst

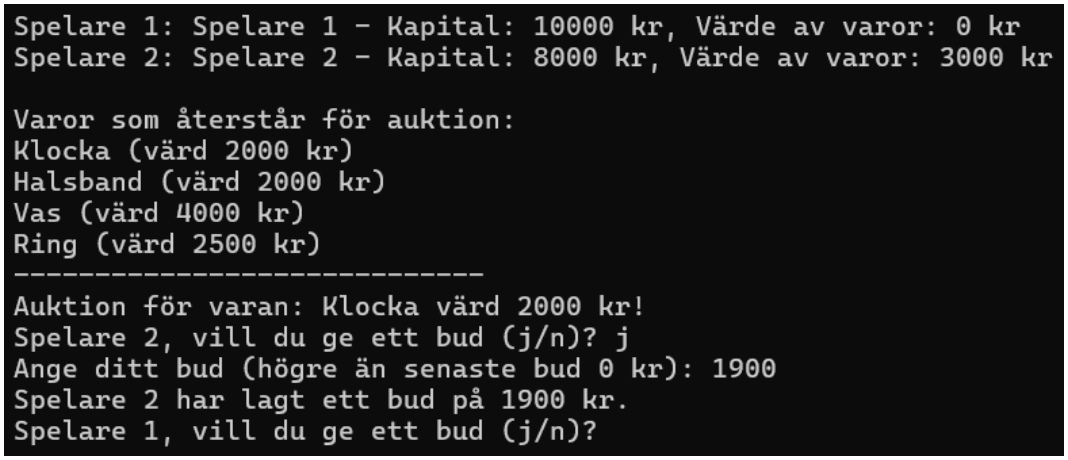
Vad programmet ska göra:

1. Be användaren mata in en lottorad med 7 unika nummer mellan 1 och 35.
2. Slumpa fram en rätt lottorad.
3. Jämför användarens lottorad med den rätta raden och räkna antal rätt.
4. Visa hur många rätt användaren fick och vilken eventuell vinst de har fått.

8.Auktionsspel.

Svårighetsgrad: Enkel nivå/Medelnivå.

Du ska implementera ett auktionsspel i C++ där två spelare deltar och budar på fem olika varor. Målet är att vinna så många och så värdefulla varor som möjligt genom att överbjuda sin motståndare, samtidigt som du håller koll på ditt kapital. Spelet avslutas när alla varor har auktionerats ut, och den spelare som har det högsta sammanlagda värdet (kapital + vunna varor) vinner spelet.



Spelregler:

1. **Startkapital:**

    - Båda spelarna börjar med 10 000 kr i kapital.
2. **Varor:**
    - Det finns 5 unika varor tillgängliga för auktion, med följande värden:
        - Klocka: 2000 kr
        - Halsband: 2000 kr
        - Tavla: 3000 kr
        - Vas: 4000 kr
        - Ring: 2500 kr
    - Varje vara auktioneras endast en gång.
3. **Auktionsomgångar:**
    - Varje omgång slumpas en vara fram som ännu inte har auktionerats ut.
    - Spelarna turas om att bjuda. Den första omgången börjar med att slumpa vem som startar.
    - Buden måste vara högre än det senaste budet och inom spelarens kvarvarande kapital.
    - Om en spelare passar eller inte kan bjuda, får den andra spelaren chansen att bjuda.
    - Om båda spelarna passar säljs inte varan till någon.
    - Vinnaren av en auktion betalar sitt bud och får varans värde till sina tillgångar.
4. **Kapital och vinster:**
    - En spelare kan aldrig bjuda mer än sitt aktuella kapital.
    - Efter varje vunnen vara minskar spelarens kapital med budbeloppet, och varans värde läggs till spelarens totala tillgångar.
5. **Spelavslutning:**
    - Spelet avslutas när alla varor har auktionerats ut.
    - Varje spelares totala kapital (kapital + värdet av vunna varor) räknas samman, och den med högst värde vinner.

Förslag på utökningar:

- Låt fler spelare delta i spelet.
- Lägg till stöd för att spelet kan sparas och återupptas senare.

- Lägg till fler varor för auktion, med olika värden och egenskaper.

Tips:

1. Använd **struct** för att hålla ordning på spelare (kapital, varuvärde) och varor (namn, värde, auktionerad eller inte).
2. Använd **statiska arrayer** för att lagra varorna och deras status.
3. Slumpa varor och startspelare med **rand()**, och initiera slumpgeneratorn med **srand(time(0))**.
4. Se till att spelarna **inte bjuder mer än sitt kapital** genom att kontrollera deras kvarvarande pengar innan bud.
5. Använd **loopar** för att repetera budgivningen tills båda spelarna har passat, och för att kontrollera om alla varor har auktionerats ut.
6. Rensa skärmen mellan varje runda med **ANSI-kod** för bättre översikt.
7. Dela upp koden i **funktioner** för att hantera budgivning, visa spelstatus och spara spelets tillstånd.
8. Implementera **filhantering** med **ofstream** och **ifstream** så att spelets status kan sparas och laddas.
9. Skriv **kommentarer** i koden för att göra programmet lättare att förstå och underhålla.

9.Perfekta tal.

***Svårighetsgrad: Medelnivå*.**

Ett **perfekt tal** är ett positivt heltal som är lika med summan av sina äkta delare (alla positiva delare förutom talet självt). Exempelvis är 6 ett perfekt tal eftersom de äkta delarna av 6 är 1, 2 och 3, och 1+2+3=6. Och 28 är ett perfekt tal eftersom dess äkta delare är 1, 2, 4, 7 och 14, och 1+2+4+7+14=28. Skriv ett C++-program som hittar de fem första **perfekta talen**.

10.Kryptering och sökning i text.

***Svårighetsgrad: Medelnivå*.**

I denna uppgift ska du skapa ett program som tar emot en textsträng från användaren, krypterar den och sparar den i en fil. Du ska också skapa en funktion som gör det möjligt att gissa ett krypterat ord och kontrollera om det finns i filen.

Del 1: Kryptera en textsträng

1. **Läs in en textsträng från användaren.**

- Låt användaren mata in en textsträng (t.ex. ett ord eller en mening).

2. **Kryptera textsträngen enligt följande metod:**
    - Om bokstaven är gemen (liten bokstav), gör om den till versal (stor bokstav).
    - Om bokstaven är versal (stor bokstav), gör om den till gemen (liten bokstav).
    - För varje bokstav, ersätt den med den bokstav som finns **tre steg längre fram i alfabetet**.
        - Exempel: "aBc" blir "DeF" (a → D, B → e, c → F).
    - Om en bokstav går förbi Z (versaler) eller z (gemener), börja om från början av alfabetet (A respektive a).
3. **Spara den krypterade textsträngen i en fil.**
    - Den krypterade textsträngen ska sparas i en fil som heter krypterade_ord.txt.

Del 2: Gissa ord i filen

1. **Låt användaren gissa ett ord.**
    - Användaren ska kunna mata in ett ord och gissa om det finns i den krypterade filen.
2. **Läs in alla krypterade ord från filen och jämför med det gissade ordet.**
    - Programmet ska svara med ett meddelande som säger om ordet finns i filen eller inte.

Exempel på inmatning och utmatning:

```
Ange textsträngen att kryptera: aBc
Den krypterade texten har sparats: DeF
Gissa ett krypterat ord: DeF
Ordet DeF finns i filen!
```

Tips:

`'isalpha(c)':` Denna funktion kontrollerar om tecknet c är en bokstav (antingen gemen eller versal). Om c är en bokstav (antingen gemen eller versal) returneras true, annars false.

`'islower(c)':` Denna funktion kontrollerar om tecknet c är en liten bokstav (gemen). Den returnerar true om det är gemen, annars false.

`'isupper(c)'`: Kontrollera om c är en stor bokstav (versal).

`'toupper(c)':` Denna funktion omvandlar tecknet c till en stor bokstav (versal), om det är en liten bokstav (gemen). Om c redan är en versal, förändras den inte.

`'tolower(c)':` Denna funktion omvandlar tecknet c till en liten bokstav (gemen), om det är en stor bokstav (versal). Om c redan är en gemen, förändras den inte.

För att använda funktionerna **`'isalpha(c)', 'islower(c)', 'isupper(c)', 'toupper(c)'`** och **`'tolower(c)'`** i C++, behöver du inkludera biblioteket **`'<cctype>'`**.

Förskjutning med alfabetet för versaler (stora bokstäver):

```
c = (c - 'A' + 3) % 26 + 'A';
```

c - 'A': Tar reda på hur långt tecknet c är från bokstaven A (index i alfabetet). Till exempel, om c = 'B', blir detta 1 (eftersom B är ett steg efter A).

+ 3: Lägg till 3 för att förskjuta tre steg framåt. Om vi hade B, blir det nu 4, vilket motsvarar E (det fjärde steget efter A).

% 26: Modulus (%) används för att hålla oss inom alfabetets 26 bokstäver. Om vi når slutet av alfabetet, börjar vi om från början. Detta gör att Z förskjuts tillbaka till början av alfabetet.

+ 'A': Vi lägger tillbaka 'A' för att omvandla resultatet till rätt bokstav i alfabetet.

Om vi har bokstaven **B**:

- **Steg 1:** c - 'A' ger 1 (eftersom B är ett steg efter A).
- **Steg 2:** Lägg till 3: 1 + 3 = 4.
- **Steg 3:** Modulus 26: 4 % 26 = 4.
- **Steg 4:** Lägg till 'A': 4 + 'A' = 'E'.
- **Resultat:** B förskjuts till E.

Förskjutning med alfabetet för gemener (små bokstäver):

```
c = (c - 'a' + 3) % 26 + 'a';
```

11. Hänga gubbe.

Svårighetsgrad: Medelnivå.

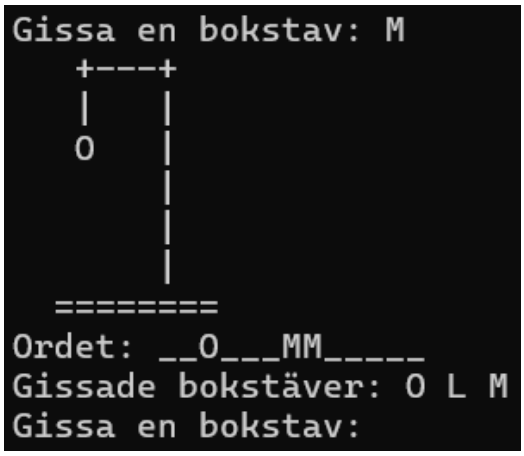
Skriv ett textbaserat program för spelet **Hänga gubbe**. Spelet ska använda ASCII-konst för att representera hänga gubbe, och information som det hemliga ordet, aktuellt läge i spelet och gissade bokstäver ska sparas i en fil.

Programmet ska också reagera om användaren gissar en bokstav som redan har gissats.

Spelets regler:

- Spelet väljer ett hemligt ord (du kan antingen välja ordet själv eller slumpa det från en lista).
- Användaren gissar en bokstav per omgång.
- Om bokstaven finns i ordet, visas den på rätt plats(er) i ordet.
- Om bokstaven inte finns i ordet, förlorar användaren ett försök.
- Användaren förlorar om den missar för många gånger (t.ex. 6 felaktiga gissningar).
- Om användaren gissar alla bokstäverna rätt innan försöken tar slut, vinner den.

ASCII-konst: Använd ASCII-konst för att rita gubben och repet vid varje felaktig gissning.



Filhantering:

- Spara det hemliga ordet, aktuell status (t.ex. hur mycket av ordet som har avslöjats), och alla gissade bokstäver i en fil.
- När användaren gissar en bokstav som redan har gissats, ska programmet varna användaren.

12. Längsta gemensamma substräng.

***Svårighetsgrad: Medelnivå*.**

Skriv ett program som tar emot två textsträngar från ett formulär och tar reda på vilken den längsta gemensamma substrängen är. En substräng är en sekvens av sammanhängande tecken som finns i båda textsträngarna.

Instruktioner:

1. **Inmatning:** Programmet ska ta emot två textsträngar från användaren.
2. **Längsta gemensamma substräng:** Programmet ska hitta den längsta gemensamma sammanhängande sekvensen av bokstäver som förekommer i båda strängarna.
3. **Utmatning:** Programmet ska skriva ut den längsta gemensamma substrängen.

Exempel: Om de två strängarna är **"banankontakt"** och **"tankbil"** är den längsta gemensamma substrängen **"ank"**.

Tips: I **<string>**-biblioteket kan **'`substr(start,length)`'** användas för att ta ut en del av en sträng (en substräng) enligt:

```
string text = "programmering";

string del = text.substr(3, 4); // "gram"
```

I **<string>**-biblioteket finns **'`find()`'** som söker efter en delsträng i en annan sträng och returnerar positionen (indexet) för den första förekomsten av delsträngen. Om delsträngen inte hittas, returnerar **'`find()`'** ett speciellt värde: **'`string::npos`'**, vilket betyder att sökningen misslyckades.

```
string text = "programmering";
string delstrang = "gram";

// Hitta positionen för delsträngen
size_t position = text.find(delstrang);  // Output: 3, eftersom "gram" börjar på index 3
```

Exempel på inmatning och utmatning:

```
Ange första textsträngen: banankontakt
Ange andra textsträngen: tankbil
Den längsta gemensamma substrängen är: ank
```

13.Simulera ett digitalt ur i C++.

***Svårighetsgrad: Enkel nivå/Medelnivå*.**

Skapat ett program som:

1. Visar tiden i formatet **"HH:MM:SS"** på skärmen, där varje del (timmar, minuter, sekunder) alltid har två siffror, även om värdet är mindre än 10.
2. Uppdaterar tiden varje sekund, som en verklig klocka.
3. Hanterar övergångar korrekt mellan sekunder, minuter och timmar.

4. Kör kontinuerligt tills programmet avslutas manuellt (Ctrl + C).

Tips:

Tidsfördröjning: Använd en funktion för att skapa en fördröjning på en sekund mellan varje uppdatering.

Visa tiden korrekt: Använd `'cout'` för att skriva ut tiden i rätt format, med hjälp av villkorssatser för att lägga till en ledande nolla när det behövs.

Om antalet timmar är mindre än 10 (t.ex. 9 eller mindre), lägger vi till en ledande nolla så att det skrivs som "09" istället för "9". Därefter skrivs själva timmen ut följt av ett kolon (:). Samma logik används för minuter och sekunder.

Så för utskrift med nollutfyllnad kan följande kod användas:

```
// Skriv ut tiden i format HH:MM:SS
if (hours < 10) cout << '0';
cout << hours << ':';

if (minutes < 10) cout << '0';
cout << minutes << ':';

if (seconds < 10) cout << '0';
cout << seconds << "\r";  // "\r" gör att tiden skrivs över på samma rad
cout.flush();  // Säkerställ att tiden uppdateras korrekt
```

Utmaningar och vidareutveckling

- Försök implementera en snabbare simulering av tiden för att kontrollera hur programmet beter sig vid snabbare tidsuppdateringar.
- Du kan lägga till funktionalitet som att låta användaren ange en starttid istället för att alltid börja från 00:00:00.
- Du kan implementera ett stoppur som räknar upp tiden från 0 när användaren startar det.

14. Stora bokstäver.

Svårighetsgrad: Medelnivå.

Skriv ett program som tar en mening från användaren och gör om varje ords första bokstav till versal (stor bokstav). Programmet ska fungera för både svenska och engelska tecken, inklusive de svenska bokstäverna å, ä och ö.

1. Be användaren att mata in en mening.

2. Dela upp meningen i ord och ändra varje ords första bokstav till en stor bokstav (versal).

3. Låt resten av bokstäverna i varje ord vara oförändrade.

4. Skriv ut meningen där alla ord börjar med en stor bokstav.

```
Enter a sentence: what a beautiful day, isn't it? let's go for a walk!
Sentence with capitalized words: What A Beautiful Day, Isn't It? Let's Go For A Walk!
```

Tips:

- Använd ’`getline()`’ för att läsa in en hel mening från användaren.
- Du kan använda funktionerna ’`toupper()`’ för att ändra en bokstav till versal.
- Tänk på att du behöver kontrollera var varje nytt ord börjar, t.ex. efter ett mellanslag.

15. En enarmad bandit (slotmaskin).

Svårighetsgrad: Medelnivå/Avancerad nivå.

Ditt uppdrag är att skapa en enkel version av en enarmad bandit (slotmaskin) i C++. I detta spel ska användaren kunna "dra i spaken" och slumpmässigt få tre symboler. Beroende på resultatet ska användaren kunna vinna eller förlora poäng. Spelet fortsätter tills användaren antingen har slut på poäng eller väljer att avsluta.

Specifikationer:

1. **Symboler**: Spelet ska ha fyra olika symboler, t ex: 7, O, C, och +. Symbolerna ska vara lagrade i en array. Dessa symboler ska visas på tre hjul varje gång användaren drar i spaken.
2. **Vinstregler**:
    - Om alla tre symboler är likadana, vinner användaren en **storvinst** och får 50 poäng.
    - Om två av tre symboler är likadana, vinner användaren en **mindre vinst** och får 20 poäng.
    - Om alla symboler är olika, förlorar användaren och inga poäng delas ut.
3. **Startkapital**: Användaren börjar med 100 poäng. Varje omgång kostar 10 poäng.
4. **Interaktion**: Användaren ska få välja om de vill fortsätta spela efter varje omgång tills kapitalet är slut eller de väljer att sluta.

Förslag på utökningar av programmet:

1. **Fler symboler**: Lägg till fler symboler i spelet (t.ex. @, *, eller #) och justera vinstreglerna för att göra spelet mer varierat.

2. **Komplexare vinstsystem**: Inför fler vinstnivåer, till exempel:
    - Om en specifik symbol (t.ex. 7) visas på alla tre hjul, ger det en extra hög vinst.
    - Skapa kombinationer där olika symboler tillsammans kan ge bonuspoäng.
3. **Grafisk representation i konsolen**: Skapa en enkel grafisk representation av hjulen med hjälp av ASCII-konst för att göra spelet visuellt mer tilltalande.

4. **Simulera rullning:**

    Fördröjning (Delay)

    I loopen kan du använda en delay för att simulera rullning.

    För att ge intrycket av att hjulen snurrar och byter symboler kan ni **rensa skärmen** mellan varje steg av simuleringen.

    Snurrande symboler

    Genom att använda moduloperatorn (%) kan du skapa en effekt där symbolerna "snurrar runt", vilket gör att varje hjul roterar bland sina symboler:

```
// Symbolerna i slotmaskinen
string symbols[] = { "7", "C", "+", "O" };

// Använd en loop för att simulera rullning av hjulen
for (int i = 0; i < 15; i++) {
    cout << "Resultat: " << symbols[(slot1 + i) % 4] << " "
                         << symbols[(slot2 + i) % 4] << " "
                         << symbols[(slot3 + i) % 4] << endl;
    this_thread::sleep_for(chrono::milliseconds(200));  // Fördröjning

    // Rensa skärmen endast för Windows
    system("cls");
}
```

    Denna metod skapar en illusion av att hjulen snurrar och byter symboler, vilket gör att programmet känns mer dynamiskt och visuellt tilltalande.

Stanna hjulen ett i taget

För att göra simuleringen ännu mer verklighetstrogen kan du låta varje hjul stanna ett i taget, istället för att alla hjulen stannar samtidigt. Börja med att stanna det första hjulet, låt det andra hjulet fortsätta snurra ett tag, och slutligen låt det tredje hjulet snurra lite längre innan det också stannar.

```
// Först stannar slot 1 (vänstra hjulet)
for (int i = 0; i < 10; i++) {
    cout << "Resultat: " << symbols[slot1] << " "  // slot 1 stannar
                         << symbols[(slot2 + i) % 4] << " "  // slot 2 snurrar
                         << symbols[(slot3 + i) % 4] << endl;  // slot 3 snurrar
    this_thread::sleep_for(chrono::milliseconds(200));
    system("cls");
}
```

Genom att låta hjulen stanna ett i taget skapar du en mer verklighetstrogen simulering av en riktig slotmaskin. Det ökar känslan av spänning när resultatet gradvis visas.

5. **Liv-system**: Istället för att enbart förlora poäng kan du introducera ett liv-system där användaren har ett antal liv, och spelet tar slut när alla liv är förbrukade.

16.Luffarschack (tic-tac-toe).

Svårighetsgrad: Medelnivå/Avancerad nivå.

Ditt uppdrag är att skapa ett enkelt **luffarschack (tic-tac-toe)** i C++ där två spelare turas om att placera sina symboler ('**x**' och '**o**') på en 3x3-spelplan**.** Lösningen ska ske med hjälp av **arrayer.** Målet är att få tre symboler i rad – horisontellt, vertikalt eller diagonalt – för att vinna spelet.

Specifikationer:

1. **Spelplan:** Spelet ska spelas på en 3x3-spelplan. Spelarna får välja vilken ruta de vill placera sin symbol i genom att ange rad och kolumn.
2. **Två spelare deltar (multiplayer):** Den ena spelaren använder symbolen '**x**', och den andra använder symbolen '**o**'.
3. **Spelomgång:** Spelarna turas om att ange vilken ruta de vill placera sin symbol i. Spelet kontrollerar att den valda rutan är ledig.
4. **Vinstvillkor:** Den första spelaren som får tre av sina symboler i rad, kolumn eller diagonal vinner spelet.

5. **Oavgjort:** Om spelplanen fylls utan att någon har vunnit, slutar spelet oavgjort.

Förslag på utökningar:

- **Utöka spelplanen** till 4x4 eller 5x5.
- Använd **ASCII-konst** för att visualisera spelplanen.
- Implementera en **datormotståndare** med spelintelligens för att spela mot användaren.

17. 4-i-rad.

***Svårighetsgrad: Medelnivå/Avancerad nivå*.**

Spelbräde:

- Rutnätet ska vara 7 kolumner brett och 6 rader högt.
- Tomma rutor representeras av symbolen ~.
- Spelare 1 har symbolen X, och spelare 2 har symbolen O.

Spelmekanik:

- Spelare 1 och 2 turas om att välja en kolumn där de vill släppa sin bricka.
- En bricka faller till den lägsta tomma platsen i den valda kolumnen.
- Om en kolumn är full kan ingen bricka läggas i den.

2. **Vinstvillkor:**
    - Spelet kontrollerar efter varje drag om någon spelare har fått fyra i rad horisontellt, vertikalt eller diagonalt.
    - Om en spelare får fyra i rad skrivs ett vinstmeddelande ut.
    - Om alla rutor är fyllda utan att någon fått fyra i rad blir det oavgjort.
3. **Skärmrensning:**
    - Rensa konsolen efter varje drag så att den senaste versionen av brädet visas.
4. **Kontrollera drag:**
    - Spelet kontrollerar att kolumnvalet är giltigt (att kolumnen inte är full och att numret är inom giltigt intervall).

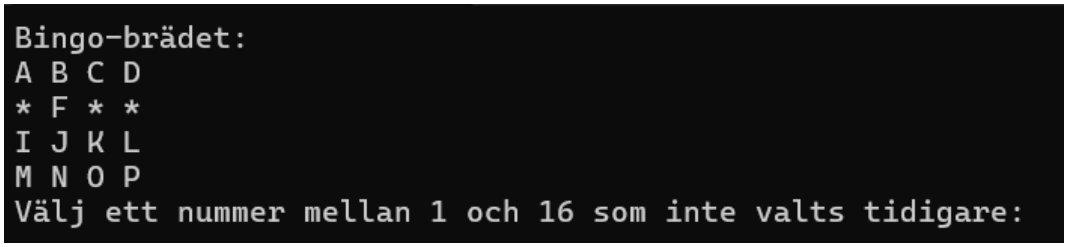
18.Bingo – variant.

***Svårighetsgrad: Avancerad nivå*.**

Du ska skapa ett **Bingo-spel** i C++ där användaren spelar på en **4x4-array.** Arrayen är fylld med slumpmässigt placerade tal från 1 till 16**,** gömda bakom bokstäverna A till P (en bokstav för varje ruta). Användaren får 16 chanser att välja ett nummer, och varje gång de väljer en ruta så avslöjas vilket tal som döljer sig bakom bokstaven. Målet är att "öppna" en hel rad, kolumn eller diagonal av rutor med valda tal.

Regler och specifikationer:

1. **Array:** Spelet består av en 4x4 array där varje ruta representeras av en bokstav från A till P. Bakom varje bokstav gömmer sig ett tal från 1 till 16, som har placerats slumpmässigt.
2. **Val av nummer:** Användaren väljer ett nummer mellan 1 och 16 för att "öppna" motsvarande ruta. Användaren får inte välja samma nummer flera gånger, och spelet ska kontrollera att samma nummer inte väljs på nytt.
3. **Vinstvillkor:** Om användaren lyckas öppna alla rutor i en hel rad, kolumn eller diagonal, vinner användaren spelet, och programmet visar meddelandet **"Bingo!"**.
4. **Spelets avslutning:** Spelet kan antingen sluta med att användaren får "Bingo" innan de gjort alla val, eller om de har valt alla rutor utan att vinna. När spelet avslutas ska användaren få en sammanfattning av sitt spel.



Förslag på utökningar:

1. **Utökade vinstkombinationer**: Implementera fler vinstkombinationer, till exempel "Två rader" eller "Ytterkanter" för att öka spelvariation.
2. **Grafisk representation**: Utöka programmet med ASCII-grafik för att visuellt visa hur brädet förändras när användaren gör sina val.
3. **Svårighetsgrad**: Låt användaren välja mellan olika svårighetsgrader, till exempel genom att ha ett större eller mindre bräde (3x3 eller 5x5).
4. **Multiplayer-läge**: Implementera ett multiplayer-läge där två spelare turas om att välja nummer och den första som får Bingo vinner.
5. **Räknare:** Implementera en räknare som håller reda på hur många försök användaren behövde för att få Bingo.

19.ASCII-labyrintspel.

Svårighetsgrad: Medelnivå/Avancerad nivå.

Din uppgift är att skapa ett enkelt labyrintspel i konsolen, där spelaren styr en agent (*****) genom en labyrint med hjälp av piltangenterna. Labyrinten består av väggar (**#**) och en utgång (**E**). Målet är att agenten ska navigera genom labyrinten utan att träffa en vägg och nå utgången.

Krav:

1. **Labyrinten**:
    - Skapa en 2D-array som representerar labyrinten.
    - **#** representerar väggar, och agenten (*****) kan inte gå igenom dessa.
    - **E** representerar utgången som agenten ska nå.
2. **Rörelse**:
    - Använd piltangenterna för att styra agenten uppåt, nedåt, vänster och höger.

- Du ska fånga tangenttryckningar och uppdatera agentens position på skärmen.
- Se till att agenten inte kan gå utanför labyrintens gränser.

3. **Skärmrensning**:
   - Rensa skärmen varje gång agenten rör sig så att bara den senaste versionen av labyrinten visas.
4. **Game Over och vinst**:
   - Om agenten går in i en vägg avslutas spelet med ett "Game Over"-meddelande.
   - Om agenten når utgången (**E**) avslutas spelet med ett "Grattis"-meddelande.

Tips:

- Använd ASCII-tecken för att skapa labyrinten, och se till att agentens position uppdateras när du trycker på piltangenterna.
- Skärmuppdatera på lämpligt sätt.

Förslag på utökningar:

1. **Flera nivåer:** Skapa flera nivåer av labyrinten med ökande svårighetsgrad.
2. **Timer:** Lägg till en timer som räknar hur lång tid det tar att lösa labyrinten.

```
#include <chrono>   // För att mäta tid
```

```
// Starta timern när spelet börjar
auto startTime = high_resolution_clock::now();
```

```
// Slut på spelet, stoppa timern
auto endTime = high_resolution_clock::now();
auto duration = duration_cast<seconds>(endTime - startTime);
```

```
// Visa hur lång tid spelet tog
std::cout << "Du spelade i " << duration.count() << " sekunder!" << std::endl;
```

3. **Begränsa antalet drag**: Spelaren får ett visst antal drag (t.ex. 60). Varje gång spelaren gör ett drag minskas detta antal. Om spelaren når 0 drag utan att ha klarat labyrinten, avslutas spelet med ett "Game Over"-meddelande. Under spelet visas hur många drag som återstår.
4. **Tidsgräns**: Du kan lägga till en tidsgräns genom att jämföra tiden som gått med en bestämd tidsgräns, och avsluta spelet om tiden tar slut.

5. **Löpande timer**: Om du vill visa tiden löpande på skärmen kan du beräkna den efter varje tangenttryckning och visa den tillsammans med labyrinten.

Förslag på reflektion: Hur får man en agent att lära sig navigera i en labyrint? Fundera över hur man kan förbättra agentens beteende så att den kan lära sig att navigera genom labyrinten. Tänk på olika strategier agenten kan använda, som att "hålla sig till en vägg", eller kartlägga labyrinten medan den rör sig.

20. Snake.

***Svårighetsgrad: Avancerad nivå*.**

Du ska skapa ett konsolbaserat Snake-spel där du styr en orm med piltangenterna. Ormen rör sig över en spelplan, och målet är att få den att äta "äpplen" som slumpas ut på planen. Varje gång ormen äter ett äpple växer den med ett segment, och spelet avslutas om ormen krockar med en vägg eller sig själv.

Spelplanen (rutnätet):

- Skapa en tvådimensionell array som representerar spelplanen.
- Använd tecknet # för att skapa väggar runtom hela spelplanen.

- Använd tecknet * för att representera ormen, som består av flera segment (starta med 2 segment).
- Använd tecknet O för att representera äpplen som slumpas ut på spelplanen.

Ormens rörelse:

- Ormen ska röra sig kontinuerligt i spelet.
- Använd piltangenterna för att ändra ormens riktning:
    - **Upp:** Piltangent upp
    - **Ner:** Piltangent ner
    - **Höger:** Piltangent höger
    - **Vänster:** Piltangent vänster

Hur du fångar piltangenter i Snake-spelet:

`_kbhit()`:

- Den här funktionen kontrollerar om en tangent har tryckts in utan att pausa spelet. Det gör att programmet kan fortsätta köra samtidigt som det väntar på ett tangenttryck.

```
if (_kbhit()) {
    // Det har tryckts in en tangent
}
```

`_getch()`:

- När vi vet att en tangent har tryckts in, använder vi `'_getch()'` för att läsa vilken tangent det är.
- För piltangenter behöver vi göra två anrop till `'_getch()'` eftersom piltangenter skickar två koder:
    - Första koden (oftast 224) indikerar att det är en piltangent.
    - Andra koden identifierar riktningen (upp, ner, vänster, höger).

```
if (_kbhit()) {  // Kontrollera om en tangent har tryckts in
    int key = _getch();  // Läs tangentens första kod
    if (key == 224) {  // Om koden är 224, är det en piltangent
        key = _getch();  // Läs riktningen
        switch (key) {
            case 72: // Upp-piltangent
                // Ändra riktning till uppåt
                break;
            case 77: // Höger-piltangent
                // Ändra riktning till höger
                break;
            case 80: // Ner-piltangent
                // Ändra riktning till nedåt
                break;
            case 75: // Vänster-piltangent
                // Ändra riktning till vänster
                break;
        }
    }
}
```

- Se till att ormen inte kan vända rakt bakåt (t.ex. om den rör sig uppåt, ska den inte kunna vända direkt ner).

Äpplen och poäng:

- Slumpa ut ett äpple (O) på en tom plats i spelplanen.
- Varje gång ormen äter ett äpple, ska den växa med ett segment och poängen ska öka med 1.
- När ormen växer, ska dess svans förlängas med ett nytt segment.

Kollision och spelavslut:

- Spelet ska avslutas om ormen krockar med en vägg eller om den krockar med sig själv.
- När spelet avslutas, ska meddelandet "Game Over!" skrivas ut, och poängen ska visas.

Skärmrensning och uppdatering:

- Använd t ex ANSI-koder för att rensa skärmen varje gång ormen rör sig så att endast den senaste versionen av spelplanen visas.

- Använd en fördröjning (0.5 sekunder) för att uppdatera ormens rörelse.

Förslag på utökningar till Snake-spelet:

1. Öka hastigheten när ormen äter ett äpple.
2. Lägg till hinder på spelplanen.
3. Skapa flera nivåer med olika layout.
4. Poänglista som sparar de 5 högsta poängen i en fil och skriver ut Highscore vid begäran.
5. Tvåspelarläge där varje spelare styr en egen orm.
6. Lägg till bonuspoäng eller specialobjekt som ger extra poäng.
7. Gör ormens färg eller utseende förändras efter ett visst antal poäng.
8. Lägg till olika typer av hinder eller faror som rör sig på spelplanen.
9. Skapa en meny för att välja olika alternativ som "Starta spel", "Visa poänglista", ”Utmaningsläge” eller "Avsluta".

21. Meningsbyggnad.

Svårighetsgrad: Medelnivå/Avancerad nivå.

I denna uppgift ska du skriva ett program som bygger slumpmässiga och grammatiskt korrekta meningar efter givna regler om hur ord från olika kategorier kan kombineras. Orden kommer från kategorier som substantiv, adjektiv, verb och artiklar och sparas i separata textfiler. Programmet ska slumpmässigt välja ord från varje kategori, böja dem korrekt beroende på artikel och om de är i singular eller plural, samt versalisera det första ordet i meningen.

Instruktioner:

1. **Skapa ordlistor som textfiler**:
   Börja med att skapa fyra textfiler som innehåller ord från varje kategori. Här är exempel på hur filerna kan se ut:

substantiv.txt

```
bil
krok
katt
hus
tåg
```

verb.txt

```
sprang
hoppade
flög
rullade
simmade
```

adjektiv.txt

```
röd
stor
snabb
blå
gul
```

artiklar.txt

```
en
ett
den
det
två
```

prepositioner.txt

```
över
under
framför
bredvid
genom
```

2. **Se till att programmet gör följande:**

- Läser in ord från filer.
- Slumpmässigt väljer ett ord från varje kategori (substantiv, verb, adjektiv, artiklar, prepositioner).
- Följer grammatiska regler, som att böja adjektiv och substantiv korrekt beroende på om artikeln är singular eller plural.
- Versaliserar det första ordet i meningen.

Exempel på utmatning:

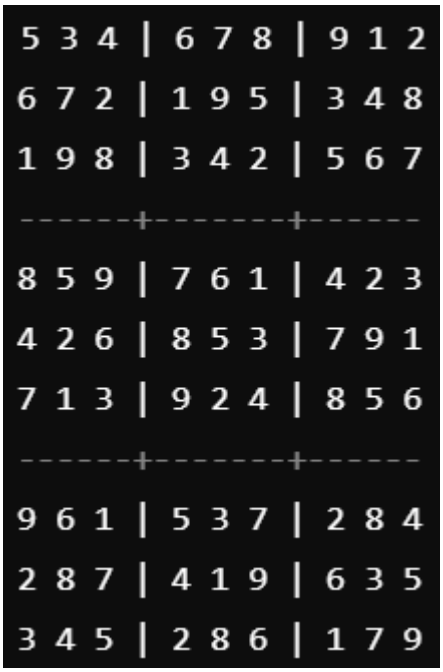
```
Slumpmässig mening: Ett tåg rullade över en röd krok.
```

22. Sudoku.

Svårighetsgrad: Medelnivå/Avancerad nivå.

I den här uppgiften ska du skriva ett program som skapar ett Sudoku-spel där användaren kan fylla i siffror i ett 9x9 rutnät. Du behöver inte implementera en algoritm som genererar Sudoku-pusslet från grunden.

Här är ett exempel på ett löst Sudoku-pussel:



Hur pusslet fungerar:

- **Varje rad** innehåller siffrorna 1 till 9 utan upprepningar.
- **Varje kolumn** innehåller också siffrorna 1 till 9 utan upprepningar.
- **Varje 3x3-sektion** innehåller siffrorna 1 till 9, utan att samma siffra förekommer två gånger inom en sektion.

Instruktioner

1.Skapa ett spelbart Sudoku-pussel i ett 9x9 rutnät där vissa siffror är förifyllda och andra är tomma. Du får använda följande fördefinierade startvärden för Sudoku-pusslet:

```
// Sudoku-pussel (ofullständigt, med 0 som tomma rutor)
int sudoku[9][9] = {
    {5, 3, 0, 0, 7, 0, 0, 0, 0},
    {6, 0, 0, 1, 9, 5, 0, 0, 0},
    {0, 9, 8, 0, 0, 0, 0, 6, 0},
    {8, 0, 0, 0, 6, 0, 0, 0, 3},
    {4, 0, 0, 8, 0, 3, 0, 0, 1},
    {7, 0, 0, 0, 2, 0, 0, 0, 6},
    {0, 6, 0, 0, 0, 0, 2, 8, 0},
    {0, 0, 0, 4, 1, 9, 0, 0, 5},
    {0, 0, 0, 0, 8, 0, 0, 7, 9}
};
```

2.Låta användaren fylla i siffror i de tomma rutorna:

- Låt användaren ange **rad, kolumn** och siffra för varje inmatning.
- Kontrollera om inmatningen är korrekt enligt Sudoku-reglerna.

- Om användaren gör en felaktig inmatning, ge ett felmeddelande och be om ett nytt värde.

3.Lagra spelets framsteg i en fil, så att användaren kan pausa spelet och fortsätta senare.

4.

Avancerad nivå: Kontrollera om inmatade värden är korrekta eller felaktiga enligt Sudoku-reglerna.

Medelnivå: Kontrollera om inmatade värden är korrekta eller felaktiga genom att jämföra varje användarinmatning med en korrekt lösning (facit).

```
// Facit (fullständigt löst Sudoku-pussel)
int solution[9][9] = {
    {5, 3, 4, 6, 7, 8, 9, 1, 2},
    {6, 7, 2, 1, 9, 5, 3, 4, 8},
    {1, 9, 8, 3, 4, 2, 5, 6, 7},
    {8, 5, 9, 7, 6, 1, 4, 2, 3},
    {4, 2, 6, 8, 5, 3, 7, 9, 1},
    {7, 1, 3, 9, 2, 4, 8, 5, 6},
    {9, 6, 1, 5, 3, 7, 2, 8, 4},
    {2, 8, 7, 4, 1, 9, 6, 3, 5},
    {3, 4, 5, 2, 8, 6, 1, 7, 9}
};
```

5.Kontrollera när spelet är färdigt (när alla rutor är korrekt ifyllda).

23. Lotto-simulering med filhantering.

Svårighetsgrad: Medelnivå/Avancerad nivå.

I denna uppgift ska du skriva ett program som simulerar ett Lotto-spel. Programmet ska slumpa fram en rätt lottorad och spara den i en fil. Därefter ska programmet slumpa fram spelade rader och rätta dem mot den rätta raden. Du ska också spara resultatet för varje spelad rad i en annan fil.

Instruktioner:

1. **Slumpa en rätt lottorad:** Programmet ska generera 7 unika nummer mellan 1 och 35 och spara dem som den rätta lottoraden i en fil.
2. **Låt användaren ange antal rader som ska spelas.** Varje rad ska också bestå av 7 unika nummer.

3. **Rätta varje rad:** Jämför varje spelad rad med den rätta raden och räkna antal rätt.
4. **Spara resultatet i en fil där varje rad innehåller:**
    - De spelade numren.
    - Antal rätt på raden.
    - Vinstsumman (beroende på antal rätt, se specifikationer nedan).

```
Rad 1: 1 3 9 21 22 24 26 - Antal rätt: 1 - Vinst: 0 SEK
Rad 2: 2 14 17 24 29 30 31 - Antal rätt: 2 - Vinst: 0 SEK
```

5. **Visa den högsta vinsten i konsolen.**

Specifikationer:

- **Antal nummer i varje rad:** 7.
- **Nummerintervall:** 1 till 35.
- **Vinster:**
    - **7 rätt:** 10 000 000 SEK
    - **6 rätt:** 500 000 SEK
    - **5 rätt:** 10 000 SEK
    - **4 rätt:** 100 SEK
    - **3 rätt:** 50 SEK
    - **Mindre än 3 rätt:** Ingen vinst

Exempel på programflöde:

1. Programmet slumpar fram en rätt lottorad och sparar den i rätt_lottorad.txt.
2. Användaren anger hur många rader hen vill spela.
3. Programmet slumpar spelade rader, rättar dem, och sparar resultatet i resultat.txt.
4. Programmet skriver ut den högsta vinsten i konsolen.

Tips:

Med hjälp av funktionen **`sort`** från **`algorithm`**-biblioteket, kan vi snabbt och enkelt sortera numren i varje rad.

```cpp
#include <iostream>
#include <algorithm>  // För sorteringsfunktionen
using namespace std;

int main() {
    // En osorterad lottorad
    int lottorad[7] = {25, 7, 34, 1, 18, 15, 12};
    // Sortera lottoraden
    sort(lottorad, lottorad + 7);  // Sortera arrayen från första till sista elementet
   return 0;
}
```

När du använder funktionen **`sort`** i C++ för att sortera en array, uppdateras arrayen direkt i minnet. Det innebär att den array som skickas in till sort kommer att vara sorterad efter att funktionen körts, utan att du behöver skapa eller returnera en ny array.

Förslag på utökningar:

1. **Spara och visa total vinst:**
    - Beräkna och visa total vinst efter alla spelade rader.
2. **Visualisering av vinster och statistik:**
    - Skapa en översikt över hur många gånger varje antal rätt (4, 5, 6, 7 rätt) inträffar när programmet körs för ett stort antal rader (t.ex. 100 000 rader). Efter simuleringen ska programmet visa både:
        - Antal gånger varje antal rätt inträffade.
        - Relativa frekvenser (hur ofta varje antal rätt inträffade jämfört med det totala antalet rader).
        - Relativa frekvenser jämförs sedan med de teoretiska sannolikheterna som vi beräknar nedan (sannolikheten för 7, 6, 5, 4 rätt).
3. **Justera vinstsummorna** baserat på det verkliga utfallet. Dela upp den totala vinstpotten mellan olika rättnivåer (t.ex. 7, 6, 5, 4, 3 rätt) beroende på hur många gånger varje antal rätt inträffar.

Sannolikheter

Sannolikheterna för att få ett visst antal rätt på Lotto kan beräknas med hjälp av kombinatorik. Det totala antalet möjliga lottorader är antalet sätt vi kan välja 7 nummer från 35 möjliga. Detta beräknas med så kallade kombinationer. **C(35,7)= 6 724 520.** $P(7\ rätt) = \frac{1}{C(35,7)} \approx 0.0000001487. P(6\ rätt) = \frac{C(7,6)\cdot C(28,1)}{C(35,7)} \approx 0.00002914. P(5\ rätt) = \frac{C(7,5)\cdot C(28,2)}{C(35,7)} \approx 0.00118. P(4\ rätt) = \frac{C(7,4)\cdot C(28,3)}{C(35,7)} \approx 0.01704. P(3\ rätt) \approx 0.10652. P(mindre\ än\ 3\ rätt) = 1 - P(3\ rätt\ eller\ fler) = 0.981751.$

24. Största talet.

***Svårighetsgrad: Avancerad nivå*.**

I den här uppgiften ska du skriva ett program som, givet en lista av positiva heltal, ordnar om dessa tal så att de bildar det största möjliga talet när de sätts samman. Exempelvis ger fältet {50, 2, 1, 9} det största möjliga talet 95021.

Tips:

Bubbel-sortering

- Bubbel-sortering är en algoritm där du går igenom listan flera gånger, jämför intilliggande element och byter plats på dem om de inte är i rätt ordning.
- Du fortsätter detta tills listan är helt sorterad.

Exempel på bubbel-sortering i denna uppgift:

- För varje par av tal i listan, t.ex. "50" och "9", jämförs de genom att du kombinerar dem på två olika sätt: "509" och "950".
- Om "950" är större än "509", byter du plats på talen så att "9" kommer före "50".

Genom att använda denna strategi kan du ordna talen så att de bildar det största möjliga talet när de sätts ihop.

Konvertera tal till strängar:

- För att jämföra två tal genom att kombinera dem som strängar, behöver du först konvertera heltal till strängar.
- Du kan använda `'sprintf()'` i `'<cstdio>'` eller `'to_string()'` med `'<string>'` för att konvertera heltal till strängar. Exempel:

```
string str = to_string(num);  // Konvertera heltal till sträng
```

Kombinera två strängar:

- För att jämföra två tal som strängar, behöver du kombinera dem i båda ordningar och jämföra resultatet.

```
string str1 = "50";
string str2 = "9";
string result = str1 + str2;  // Kombinera strängarna
```

Jämföra två kombinerade tal:

- När du har kombinerat två tal som strängar, kan du jämföra dem med **>**-operatorn om du använder **string**-typen. Du **behöver inte använda `'stoi()'`** för att jämföra storleken på strängarna. Strängjämförelser fungerar direkt i C++ och använder lexikografisk ordning.

  Hursomhelst, visar följande exempel hur `'stoi()'` som kräver paketet `'<string>'` fungerar: `'int num=stoi("12345");'` omvandlar strängen "12345" till ett heltal (int) som lagras i heltalsvariabeln num.

Byta plats på element:

- När du jämför två tal i din bubbelsorteringsalgoritm, kan du byta plats på två element genom att använda en temporär variabel.

```
string temp = strNumbers[i];
strNumbers[i] = strNumbers[j];
strNumbers[j] = temp;
```

25.Ping Pong.

Svårighetsgrad: Avancerad nivå.

I den här uppgiften ska du skapa ett textbaserat Ping Pong-spel där två spelare styr var sitt racket i ett rutnät. Bollen rör sig automatiskt, och målet för varje spelare är att reflektera bollen och samla poäng. Spelet påminner om det klassiska Ping Pong, men i en enklare form.

```
###############
#     ***     #
#             #
#             #
#           O #
#             #
#             #
#             #
#             #
#             #
#             #
#             #
#             #
#    ***      #
###############
Spelare 1 poäng: 0 | Spelare 2 poäng: 1
```

Specifikationer:

1. **Rutnätet:**
    - Rutnätet är 15x15 tecken stort och har väggar på ytterkanterna. Väggarna visas med symbolen #.
    - Spelplanen ska se ut som en ram av väggar runt bollen och racketarna.
2. **Spelarnas racketar:**
    - Racketarna består av tre tecken breda segment markerade med *.
    - Spelare 1:s racket ska finnas på andra raden (överst) och styrs med vänster- och högerpilarna.
    - Spelare 2:s racket ska finnas på näst sista raden (nederst) och styrs med tangenterna 'a' (vänster) och 'd' (höger).
3. **Bollen:**
    - Bollen representeras med symbolen O och börjar någonstans i rutnätet.
    - Bollen rör sig automatiskt i riktning mot en av spelarna och kan reflekteras när den träffar racketen eller en sidovägg.
    - Om bollen träffar mitten av racketen rör den sig rakt tillbaka. Om den träffar kanten på racketen, reflekteras den diagonalt.
    - När bollen träffar översta eller nedersta väggen utan att bli reflekterad, får motståndaren ett poäng och spelet fortsätter från samma plats.
4. **Poängräkning:**
    - När en spelare missar bollen och den träffar spelarens vägg, får motståndaren ett poäng.
    - Spelet fortsätter och bollen ändrar bara riktning när poäng ges, den startar inte om från mitten.

Förslag på utökningar:

1.Startmeny där spelarna kan:

- Ange sina namn.
- Välja svårighetsgrad, vilket påverkar hastigheten på spelet (genom att ändra delay mellan varje uppdatering).

2.Textmeddelanden när:

- En spelare vinner.
- Spelet startar om efter en omgång.

26.Sänka skepp (Battleships).

***Svårighetsgrad: Avancerad nivå*.**

Ditt mål är att sänka datorns alla fartyg innan den sänker dina. Du och datorn har varsin spelplan med en flotta som består av olika fartyg. Genom att skjuta på varandras spelplaner, försöker ni träffa och sänka varandras fartyg.

Spelplan och fartyg:

- Spelplanen är en 10x10 grid, där varje ruta kan innehålla vatten eller ett fartyg.
- Varje spelare (både du och datorn) har följande fartyg:
    - **1 Slagskepp** (4 rutor långt)
    - **1 Kryssare** (3 rutor långt)
    - **1 Torpedbåt** (2 rutor långt)
    - **1 U-båt** (1 ruta långt)

Hur du placerar dina fartyg:

1. Du kommer att få placera dina fartyg ett i taget på din spelplan.
2. För varje fartyg anger du:
    - **Rad** (0-9) där fartyget ska börja.
    - **Kolumn** (0-9) där fartyget ska börja.
    - **Riktning** (H för horisontellt eller V för vertikalt).
3. Se till att dina fartyg inte överlappar varandra och att de inte går utanför spelplanen.

Spelets gång:

1. **Placera dina fartyg** på spelplanen enligt instruktionerna ovan.
2. **Skjut på datorns spelplan**:
    - Ange en rad och en kolumn där du tror att ett av datorns fartyg kan finnas.
    - Om du träffar ett fartyg markeras rutan som **träff**.
    - Om du missar markeras rutan som **miss**.
    - Du får veta om det blev en träff eller en miss.

~: Vatten, ingen träff.
O: Fartyg.
X: Träffad del av fartyget.
.: Missad skott.

3. **Datorn skjuter** på din spelplan:
    - Datorn gissar också på en rad och en kolumn för att försöka sänka dina fartyg.
    - Datorn meddelar om den träffar eller missar.
4. **Fortsätt spela**:
    - Turas om att skjuta tills en av er har sänkt alla motståndarens fartyg.

Vinst och förlust:

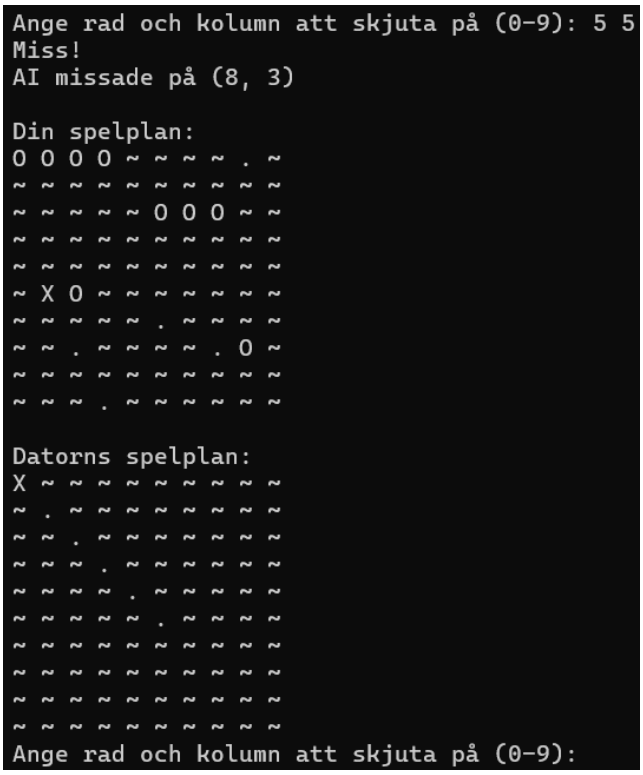
- **Du vinner** spelet om du lyckas sänka alla datorns fartyg innan den sänker dina.
- **Du förlorar** om datorn sänker alla dina fartyg innan du sänker datorns.

Spara och ladda spelet:

- **Spara spelet**: Om du vill pausa spelet och fortsätta senare, kan du spara spelet. Detta sparar båda spelplanerna (ditt och datorns).
- **Ladda spelet**: När du återvänder till spelet kan du ladda spelet från den sparade filen och fortsätta där du slutade.

Använda spelets meny:

- **Spela**: Starta en ny runda och fortsätt spela.
- **Spara**: Spara spelet så att du kan återuppta det senare.
- **Ladda**: Ladda ett tidigare sparat spel och fortsätt därifrån.
- **Avsluta**: Avsluta spelet.



Förslag på utökningar:

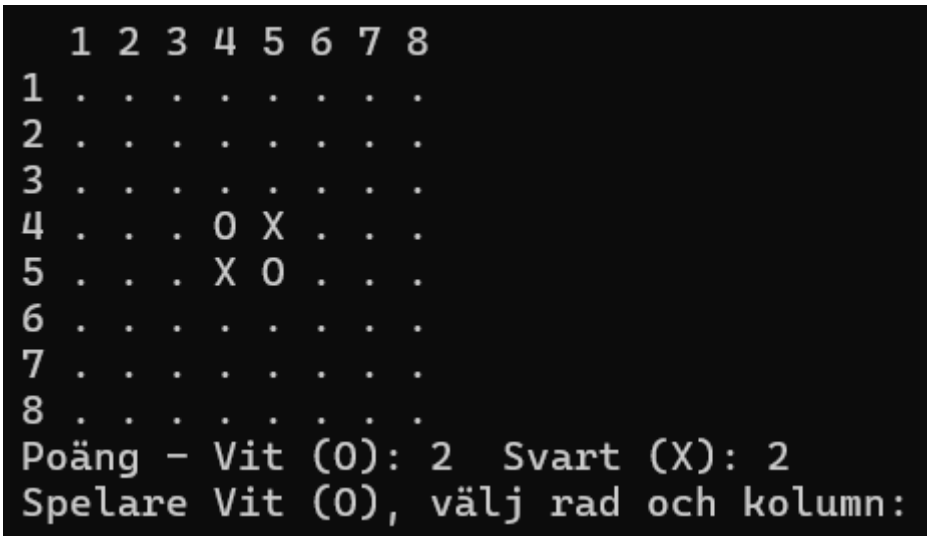
1. Lägg till fler fartygsklasser eller olika storlekar på spelplanen.
2. Skapa en AI-spelare som gör drag enligt strategin Hitta och förfölja: Skjuter slumpmässigt tills den hittar en träff, och därefter försöker den skjuta runt träffen för att sänka fartyget.

27. Othello.

***Svårighetsgrad: Avancerad nivå*.**

Othello (kallas även **Reversi)** är ett klassiskt brädspel som spelas på en 8x8-spelplan med brickor som har en vit sida och en svart sida. Spelet går ut på att vända motståndarens brickor så att de får den egna spelarens färg, genom att placera sina brickor på spelplanen.

Spelet avslutas när spelplanen är full eller när ingen av spelarna kan göra fler drag. Den spelare som har flest brickor i sin färg på spelplanen när spelet är slut vinner. Du ska skapa ett program i C++ där två spelare turas om att spela Othello mot varandra. Programmet ska följa spelets regler och visa aktuell spelplan samt poängställning under spelets gång.



Regler:

- Spelet spelas på en 8x8-spelplan.
- Spelet startar med fyra brickor i mitten av spelplanen: två vita och två ”svarta”, placerade diagonalt mot varandra.
- En spelare spelar med vita brickor (O) och den andra med ”svarta” brickor (X).
- Spelarna turas om att lägga en bricka på spelplanen.
    - När en bricka läggs, måste den placeras så att en eller flera av motståndarens brickor "fångas" mellan den nya brickan och en annan bricka av samma färg i en rak linje (horisontellt, vertikalt eller diagonalt).
    - Alla fångade brickor vänds till den färg som lades ut.
- Om en spelare inte kan göra ett giltigt drag (dvs. omsluta minst en av motståndarens brickor), måste denne stå över sitt drag.
- Spelet är slut när spelplanen är full, eller när båda spelarna inte kan göra fler drag.
- Vinnaren är den spelare som har flest brickor i sin färg på spelplanen när spelet avslutas.

Specifikationer:

1. **Spelplan och brickor:**
    - Spelplanen ska vara en 8x8-array med tomma platser (.), vita brickor (O), och ”svarta” brickor (X).
    - Programmet ska initialisera spelplanen med fyra brickor i mitten.
2. **Spelets gång:**

- Två spelare turas om att spela.
- Vid varje drag ska programmet kontrollera om spelaren kan vända motståndarens brickor. Om draget är giltigt, ska brickorna vändas.
- Om spelaren inte kan göra ett giltigt drag, ska spelaren stå över sitt drag och nästa spelare får sin tur.

3. **Poängräkning:**
    - Efter varje drag ska programmet visa spelplanen och poängställningen (antal brickor för varje spelare).
    - När spelet är slut ska det programmet meddela vem som vann (eller om det blev oavgjort).
4. **Spelslut:**
    - Spelet avslutas när spelplanen är full eller när ingen av spelarna kan göra fler drag.
    - Programmet ska meddela den slutliga poängen och utse vinnaren.

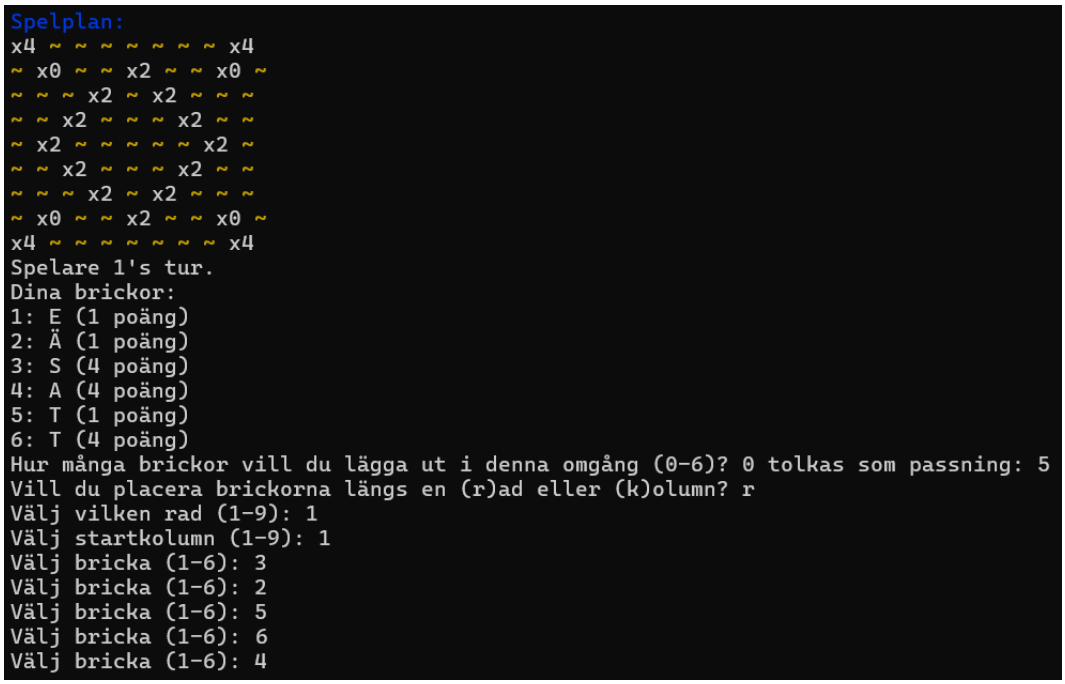
Förslag på utökningar:

- Implementera fler avancerade regler, t.ex. att spelet visar möjliga drag för spelarna.
- Implementera ett AI som spelar mot människan och försöker göra smarta drag.
- Lägg till färger i utskriften för att tydligare visa skillnad mellan vita och ”svarta” brickor.

28. Alfapet-variant.

***Svårighetsgrad: Avancerad nivå*.**

Du ska skriva ett program i C++ som simulerar ett Alfapet-liknande spel för två spelare. Spelet består av en 9x9-spelplan där spelarna turas om att lägga ut bokstavsbrickor för att bilda giltiga ord. Bokstäverna dras från en gemensam bokstavspåse, där bokstavsfördelningen har anpassats för att skapa en bättre spelupplevelse.



Bokstavsfördelning för en bättre spelupplevelse

För att spelet ska kännas balanserat och ge möjlighet till att bilda många ord, har bokstäverna i påsen en viss fördelning baserad på deras frekvens i språket. Vi använder en total mängd på 95 bokstäver i påsen, där varje bokstav förekommer enligt följande proportioner:

- **E** förekommer 12 gånger, **A** 9 gånger, **N** 8 gånger, **S** 8 gånger, **R** 7 gånger, **T** 7 gånger, **O** 6 gånger, **L** 5 gånger, **D** 4 gånger, **M** 4 gånger, **G** 3 gånger, **K** 3 gånger, **V** 3 gånger, **F** 2 gånger, **I** 2 gånger, **H** 2 gånger, **Å** 2 gånger, **Ä** 2 gånger, **Ö** 2 gånger, och **Z**, **X**, **J**, **Y** förekommer 1 gång vardera.

Denna fördelning ger en bättre balans av konsonanter och vokaler, vilket gör spelet roligare och enklare att bilda ord.

Grundläggande regler för spelet

1. **Spelplan**: Består av ett 9x9-rutnät där tomma rutor visas med symbolen "~". Vissa rutor har speciella poängmultiplikatorer:
    - Rutor markerade med "x4" multiplicerar bokstavens poäng med 4.
    - Rutor markerade med "x2" multiplicerar bokstavens poäng med 2.
    - Rutor markerade med "x0" ger inga poäng för den bokstaven.
2. **Bokstavspåse**: Bokstäverna dras från en gemensam påse med 95 bokstäver, där fördelningen följer den frekvens som beskrivs ovan.

3. **Brickställ**: Varje spelare har ett brickställ med 6 slumpmässigt utdelade bokstavsbrickor. Efter varje drag fylls brickstället på till 6 brickor från bokstavspåsen, så länge det finns bokstäver kvar i påsen.
4. **Spelets gång**:
    - Spelarna turas om att lägga ut bokstäver på spelplanen. På varje tur får spelaren välja att lägga ut 0-6 bokstäver på spelplanen.
    - Spelaren kan välja att lägga ut bokstäver längs en rad eller en kolumn. Bokstäverna måste placeras på tomma rutor, och spelaren måste välja både startposition och hur många brickor som ska läggas ut.
    - Efter varje drag räknas poängen baserat på bokstävernas poängvärde och eventuella multiplikatorer på rutor där bokstäverna placeras.
    - Om spelaren inte vill lägga ut några brickor kan hen välja att passa.
    - När en spelare lägger ut sina brickor kontrolleras om det bildas giltiga ord både längs raden eller kolumnen, samt längs eventuella andra rader eller kolumner som påverkas. Använd en ordlista (t.ex. ordlista.txt) för att kontrollera giltiga ord.
5. **Spelets slut**:
    - Spelet avslutas när något av följande inträffar:
        - 81 bokstäver har lagts ut på spelplanen (spelplanen är full).
        - Båda spelarna har valt att passa två gånger i rad.
    - Den spelare som har högst poäng när spelet avslutas vinner.

Förslag på utökningar:

- Utöka spelet med fler specialrutor (som x3-poäng).
- Lägg till fler poängberäkningsfunktioner (som att multiplicera ordvärdet istället för bokstavsvärdet).
- Låt spelaren spara spelet i en fil och återuppta det senare.
- Låt användaren få välja mellan olika spelplaner.
- Lägg till en startmeny där spelarna också kan ange sina namn.

29. Textbaserat schackspel.

***Svårighetsgrad: Avancerad nivå*.**

Du ska utveckla ett **textbaserat schackspel** där två spelare turas om att göra drag i samma fönster. Spelet ska följa de grundläggande reglerna för schack och ha funktioner för giltiga drag, schack, remi och schackmatt. Fokus ligger på noggrann felhantering, tydlig presentation av spelbrädet och korrekt spelmekanik.

1. Spelplanen

- **Bräde:** 8x8 rutor, markerade med bokstäver (a till h) för kolumner och siffror (1 till 8) för rader.
- **Startuppställning:** Standarduppställning för schack.
- **Färger:**
    - **Vita pjäser:** Vita textsymboler
    - **Svarta pjäser:** Blå textsymboler
- **Symboler för pjäser:**
    - **B:** Bonde
    - **T:** Torn
    - **S:** Springare
    - **L:** Löpare
    - **D:** Dam
    - **K:** Kung

Exempel på startbräde:

2. Spelmekanik

1. **Turordning:** Vit börjar alltid. Spelarna turas om att göra drag.
2. **Inmatning av drag:**
    - Spelaren anger vilket fält de vill flytta **från** (ex: e2).
    - Spelaren anger vilket fält de vill flytta **till** (ex: e4).
    - Programmet detekterar själv vilken pjäs som (eventuellt) berörs.
3. **Kontroller:**
    - Är startfältet korrekt? (Finns spelarens pjäs där?)
    - Är draget giltigt enligt pjäsens regler?
    - Leder draget till att den egna kungen hamnar i schack?
4. **Felhantering:**
    - Ogiltiga drag ger ett felmeddelande, och spelaren får ange ett nytt drag.

3. Schack, schackmatt, remi och vinstvillkor

3.1 Schack

- Programmet ska varna spelaren om deras kung står i schack efter ett drag.
- En spelare får inte göra ett drag som lämnar den egna kungen i schack.

3.2 Schackmatt

- Spelet avslutas om en spelare är i schack och inte kan göra något lagligt drag för att undvika hotet.
- Programmet meddelar tydligt att spelet är slut och vem som har vunnit.

3.3 Uppgivelse

- En spelare kan frivilligt ge upp genom att ange ett specifikt kommando (t.ex. resign).
- Programmet meddelar tydligt att spelaren har gett upp och vem som har vunnit.

3.4 Flaggfall (Tidsförlust)

- Om spelet inkluderar en tidsbegränsning per drag eller per match:
    - En spelare förlorar om deras tid löper ut.

    - För att vinna på tid måste den vinnande spelaren ha tillräckligt med material för att sätta matt.

- Programmet ska tydligt meddela om en spelare har förlorat på tid.

3.5 Remi (Oavgjort)

Spelet kan sluta oavgjort på följande sätt:

- **Patt:** Den spelare som är i tur kan inte göra något giltigt drag och deras kung är inte i schack.
- **Upprepning av ställning:** Om samma position uppstår tre gånger med samma spelare i tur och samma möjliga drag. Vanligtvis sker detta när en spelare kan utföra evig schackning.
- **50-dragsregeln:** Om inga bönder har flyttats och ingen pjäs har slagits under de senaste 50 dragen.
- **Otillräckligt material:** Om båda spelarna har för lite material för att kunna sätta schackmatt (t.ex. kung mot kung eller kung och löpare mot kung).

Exempelmeddelanden vid avslut:

- ”Vit vinner genom schackmatt! ”
- ”Svart vinner genom uppgivelse! ”
- ”Vit vinner genom flaggfall! ”
- ”Spelet slutar remi genom patt! ”
- ”Spelet slutar remi genom 50-dragsregeln! ”

4. Lagring i textfiler

1. **Draglogg:** Alla drag sparas i en textfil (ex: draglogg.txt) i format:

```
Vit: e2 -> e4
Svart: e7 -> e5
```

2. **Brädeposition:** Efter varje drag ska brädets nuvarande position sparas i en textfil (ex: spelbräde.txt) så att spelet kan återupptas senare.

5. Grafisk representation i terminalen

- Spelbrädet ska visas tydligt efter varje drag.
- Rader (1-8) och kolumner (a-h) ska vara numrerade.
- Vita pjäser ska visas i **vit text** och svarta pjäser i **blå text**.

6. Extra utmaningar (Valfritt)

Du kan välja att implementera en eller båda av följande utmaningar:

1. **Tidsbegränsning per drag:** Inför en timer där varje spelare har en viss tid (ex: 30 sekunder) att göra sitt drag.
2. **Tidsbegränsning per match:** Inför en total tidsbegränsning för hela partiet (ex: 10 minuter per spelare).

7. Felhantering och robusthet

- Programmet ska hantera felaktiga inmatningar utan att krascha.
- Det ska inte gå att göra drag som sätter den egna kungen i schack.
- Schack, schackmatt, remi, uppgivelse och flaggfall ska identifieras korrekt.

Tips och resurser

- Läs mer om schackregler på spelregler.org/schack.
- Planera ditt program noggrant innan du börjar koda.
- **Grundstruktur för programmet**

  Ett textbaserat schackspel kan struktureras enligt följande huvudkomponenter:

  - **Spelbräde:** Representera brädet med en 2D-array eller vektor.
  - **Pjäser och regler:** Implementera regler för varje pjäs (rörelse och slag).
  - **Spelmekanik:** Hantera turordning, inmatning av drag och validering av drag.
  - **Schack, schackmatt, remi och vinstvillkor:** Kontrollera och hantera dessa tillstånd.
  - **Lagring:** Spara drag och brädets tillstånd i textfiler.
  - **Terminalpresentation:** Visa brädet och statusmeddelanden.
  - **Felhantering:** Grundläggande validering av användarens inmatning.
  - **Extra funktioner:** Tidsbegränsningar.

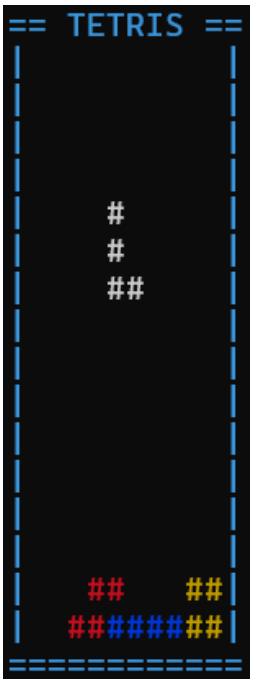
30. Textbaserat Tetris.

Svårighetsgrad: Avancerad nivå.

Du ska utveckla ett **textbaserat Tetris-spel i C++** där spelet körs direkt i terminalen. Spelet ska följa de grundläggande reglerna för Tetris och innehålla funktioner för styrning, kollisionskontroll, ”gravitation” och poängräkning.

1. Spelplanen

- **Storlek:** 10 kolumner × 16 rader.
- **Design:**
  - **Tak och golv:** Cyan färg.
  - **Sidoväggar:** Cyan färg.
  - **Tetrominos:** Varje typ av Tetromino ska ha en egen färg.
- **Symboler:**
  - **Tom ruta:** .
  - **Blockruta:** #.



1.1 Tetrominos

- Spelet ska innehålla minst **fyra olika Tetrominos**:
  1. **O (Gul)**
  2. **I (Blå)**
  3. **S (Röd)**
  4. **L (Vit)**
- Tetrominos ska kunna:
  - **Falla nedåt automatiskt** (”gravitation”).
  - **Styras med piltangenterna** (vänster, höger, ner).

    - **Roteras med en tangent** (ex. upp).
    - **Låsas på plats när de når botten eller landar på en annan Tetromino.**

1.2 Spelmekanik

- **”Gravitation”:** Tetrominos ska automatiskt falla stegvis.
- **Kollision:** Tetrominos får inte gå utanför spelplanen eller överlappa andra block.
- **Fyllda rader:** När en rad är helt fylld ska den rensas och alla ovanstående block ska flyttas ner.
- **Game Over:** Om en inkommande Tetromino inte får plats på spelplanen är spelet över.

2. Styrning

Använd följande tangenter för att styra Tetrominos:

- **← (Vänsterpil):** Flytta Tetromino åt vänster.
- **→ (Högerpil):** Flytta Tetromino åt höger.
- **↓ (Nedåtpil):** Snabbare nedfärd.
- **↑ (uppåtpil):** Rotera Tetromino.
- **Q:** Avsluta spelet.

3.Funktionella krav

1. **Ritning av spelplanen och Tetrominos:**
    - Tetrominos ska vara **synliga under hela fallet**.
2. **Kollisionskontroll:**
    - Förhindra Tetrominos från att lämna spelplanen.
    - Förhindra överlappning med andra block.
3. **”Gravitation”:**
    - Tetrominos ska falla automatiskt med en fördröjning (ex. 0.5 sekunder).
4. **Rotation:**
    - Tetrominos ska kunna roteras med en tangent.
5. **Poängräkning:**
    - En rad ger poäng när den fylls och tas bort.

4.Tekniska krav

- Använd ANSI-färgkoder för att färglägga spelplanen och Tetrominos.
- Använd **2D-arrayer eller vektorer** för att representera spelplanen.
- Hantera **kollisioner och rad-rensningar korrekt**.
- Skriv **funktioner för varje delmoment** (t.ex. moveTetromino(), rotateTetromino(), isCollision(), clearLines()).

5.Extra utmaningar (Valfritt)

- **Sparfunktion för att pausa och återuppta spelet**

    - Låt spelaren välja att pausa spelet och spara det nuvarande tillståndet till en fil (ex. savegame.txt).
    - Implementera en funktion för att ladda det sparade spelet när spelaren startar om spelet.
    - Att ladda sparat spel kan vara val i huvudmenyn innan spelet startar.

- **Highscore-lista (Topp 5)**

    - Skapa en highscore-fil (highscores.txt) där de 5 bästa poängen och spelarnamn sparas.
    - Att visa highscore-listan och registrera spelarnamn kan vara alternativ i huvudmenyn innan spelet startar.
- **Svårighetsgrad (”Gravitationens styrka” ↔ fallhastighet)**
    - Låt spelaren välja mellan **tre olika svårighetsgrader** i huvudmenyn:

        1. Lätt: 1 sekund per fallsteg.

        2. Medel: 0.5 sekunder per fallsteg.

        3. Svår: 0.2 sekunder per fallsteg.

    - Justera spelhastigheten baserat på den valda nivån.

31. Alltid 100.

Svårighetsgrad: Medelnivå/Avancerad nivå.

Skriv ett program som visar alla möjligheter att kombinera talen 1, 2, 3... 9 (i den ordningen) med + och - så att resultatet blir 100.

- *Exempel*: 1 + 2 + 34 - 5 + 67 - 8 + 9 = 100

32. Poker (Texas Hold 'em).

Skriv ett program som simulerar några olika moment i pokerspelet Texas Hold'em:

1. Utdelning av två slumpade kort till sex spelare.
2. Tre av de återstående korten läggs ut på bordet.
3. Alla händer kontrolleras och den bästa handen visas.

4. Det fjärde kortet läggs ut och punkt 3 görs igen.
5. Det femte kortet läggs ut och punkt 3 görs igen.
6. Bygg på med satsningar om du hinner.
7. Spara alla händer och utdelade kort i filer.

Länk till spelregler:

https://www.spelregler.org/texas-holdem-regler/