Part III: Spillet 1024

Maksimal score for del 3 er 200 poeng.

Introduksjon til spillet 1024

Spillet `1024` ble veldig populært i flere utgaver og kloner i år 2014. Det går ut på at man har et spillbrett med 4x4 ruter, der to av rutene til å begynne med har verdien 2. Oppgaven til spilleren er å få en av rutene i brettet til å inneholde verdien 1024. En spiller kan utføre fire forskjellige trekk, skyve alle brikker oppover, til høyre, nedover eller til venstre. Når brikkene skyves til en av sidene vil to like brikker som ligger langs aksen det flyttes legges sammen og bli en ny brikke med summen av de opprinnelige. Hvis du ikke er kjent med spillet kan du prøve en utgave av det på https://poweroftwo.nemoidstudio.com/1024.

Vi har allerede implementert en grafisk representasjon, men spillets logikk mangler. I denne eksamensoppgaven skal du implementere stort sett all logikk for spillet over flere deloppgaver.

Hvordan besvare del 3?

Alle oppgavene i del 3 er satt opp slik at de skal besvares i filen `Game.cpp`. Hver oppgave har en tilhørende unik kode for å gjøre det lettere å finne frem til hvor i filen du skal skrive svaret. Koden er på formatet `<tegn><siffer> (TS)`, eksempelvis `G1` og `G2`. I `Game.cpp` vil du for hver oppgave finne to kommentarer som definerer henholdsvis begynnelsen og slutten av koden du skal føre inn. Kommentarene er på formatet:

`// BEGIN: TS` og `//END: TS`.

Det er veldig viktig at alle svarene dine er skrevet mellom slike kommentar-par, for å støtte sensurmekanikken vår. Hvis det allerede er skrevet noen kode mellom BEGIN- og END-kommentarene i filene du har fått utdelt, så kan, og ofte bør, du erstatte den koden med din egen implementasjon.

For eksempel, for oppgave G1 ser du følgende kode i utdelte `Game.cpp`

```
void Game::new_game() {
    // BEGIN: G1
    // END: G1
}
```

Etter at du har implementert din løsning, bør du ende opp med følgende istedenfor

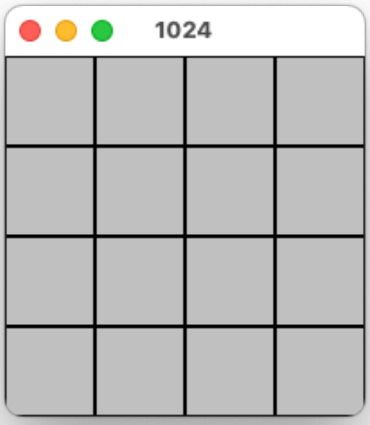
```
void Game::new_game() {
    // BEGIN: G1

    /* Your answer here. Code and // possibly explaining comments */

    // END: G1
}
```

Merk at BEGIN- og END-kommentarene **IKKE skal fjernes**.

Til slutt, hvis du synes noen av oppgavene er uklare, oppgi hvordan du tolker dem og de antagelsene du må gjøre som kommentarer i den koden du sender inn.

I `zip`-filen finner du bl.a. `Game.h` og `Game.cpp`. Det er kun `Game.cpp` som skal redigeres for å komme i mål med oppgavene. Du kan se i headerfilene for å få en oversikt over deler av koden før du starter å løse oppgavene. `Game.h` inneholder klassedefinisjonen som tar for seg spillets logikk og er sammen med `Game.cpp` filene som er viktige i denne eksamensoppgaven. `Gamewindow.h`, `Tile.h`, `settings.h` og `utilities.h` inneholder diverse deklarasjoner og definisjoner som brukes til den grafiske representasjonen - det er ikke nødvendig å sette seg inn i eller forstå koden som er knyttet til grafikk i denne eksamenens del 3.

Før du starter må du sjekke at den (umodifiserte) utdelte koden kjører uten problemer. Du skal se det samme vinduet som i figur 1. Når du har sjekket at alt fungerer som det skal er du klar til å starte programmering av svarene dine.



Figur 1: Utlevert kode uten endringer.

`Game` - Spillogikk (200 poeng)

Spillets logikk skal implementeres i klassen `Game`. Den skal kun inneholde spillets logikk og har ansvar for å korrekt flytte brikkene på brettet når spilleren velger å bevege dem opp, ned, til høyre eller venstre, samt starte spillet på nytt.

Spillbrettet er representert av 1D-vectoren `vector<int> board`, som inneholder board_size $*$ board_size antall heltall. Du vil implementere en 2D-indeksering av denne vectoren for å gjøre det lettere å implementere logikken, men også tillate brukere av klassen å indeksere et rutenett.

Spilleren kan velge å skyve alle brikker til høyre, venstre, opp eller ned. Siden alle operasjoner i utgangspunktet er like, men gjøres i forskjellige retninger skal vi først implementere skyving av alle brikker til høyre og gjenbruke den metoden for å skyve brikkene i de andre retningene. Som med alle andre programmer er det mulig å implementere dette spillet på mange forskjellige måter og i denne oppgaven skal vi bl.a. bruke matriseoperasjoner til å manipulere spillbrettet så vi kan gjenbruke høyreskyvingen. Mer om dette når det er relevant i hver enkelt oppgave.

Vi har overlastet `operator<<(ostream&, const Game&)`, som du kan bruke for å skrive ut spillbrettet til terminalen når du måtte ønske. Merk at funksjonen ikke vil fungere før etter oppgave `G4` siden funksjonene som henter verdiene fra spillbrettet mangler implementasjonsdetaljer. Du får også utdelt diverse funksjoner som kan brukes til å debugge logikken underveis, se oppgave `G6` og tabell 1 for mer informasjon.

Oppgavene må ikke gjøres i en bestemt rekkefølge. F.eks. kan funksjonen du implementerer i av deloppgave G5 gjenbrukes i tidligere oppgaver hvis du ønsker det. Du står fritt til å bruke metoder definert hvor som helst i del 3 andre steder i del 3.

1. (10 points) **G1:** `Game::index` **- Beregn 1D-index ut fra 2D-koordinaterBeregn 1D-index ut frå 2D-koordinater**

   Funksjonen skal returnere en verdi som tilsvarer 1D-indeksen beregnet fra 2D-koordinatene til elementet med koordinatene (x, y) i spillbrettet. F.eks. skal $(0, 0)$ gi 0 og $(3, 3)$ gi 15 når spillbrettet er $4x4$ stort. Bruk gjerne medlemsvariabelen `board_size`, som inneholder størrelsen på spillbrettet langs en akse, for eksamensoppgaven er det 4 (spillbrettet er 4x4).

2. (10 points) **G2:** `int Game::at(int x, int y) const` **- Les en verdi fra 2D-koordinat**

3. (10 points) **G3:** `int& Game::at(int x, int y)` **- Les en verdi fra 2D-koordinat**

   Funksjonene skal returnere verdien som befinner seg i spillbrettets posisjonen (x, y). Merk at det er *to* medlemsfunksjoner som skal implementeres (G2 og G3), `int Game::at(int x, int y) const` og `int& Game::at(int x, int y)`. Spillets brikker er lagret i medlemsvariabelen `board`.

   Hvis posisjonen i spillbrettet ikke eksisterer skal funksjonen kaste et passende unntak av typen `std::out_of_range`.

   Vi implementerer to funksjoner her for å ha mulighet til å hente både referanser til verdier i spillbrettet og kopier av verdiene. Det er hensiktsmessig siden vi ønsker begge funksjonalitetene, 1) for å gjøre det lettere å overskrive verdier i spillbrettet og 2) for å lese verdier uten mulighet til å redigere spillbrettet (f.eks. en spiller som ser spillbrettet skal ikke ha mulighet til å manipulere brikkene, det ville vært en enkel måte å jukse i spillet på).

4. (10 points) **G4:** `Game::new_game()` **- Nytt spill**

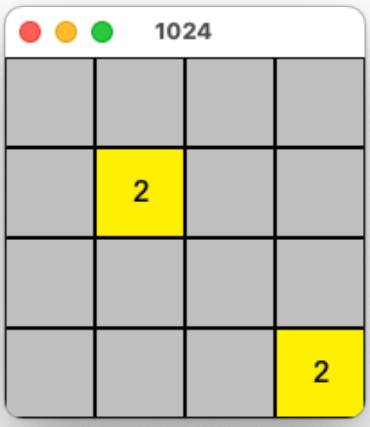
   Medlemsfunksjonen skal gjenstarte spillet så en spiller kan begynne med blanke ark. Spillbrettet skal kun bestå av 0-verdier, med unntak av to fliser som er tilfeldig plassert på spillbrettet. Se figur 2 for et eksempel på hvordan spillbrettet kan se ut etter denne funksjonen.

   Merk at du skal bruke medlemsfunksjonen `place_new_2` til å plassere 2-tallene - se neste deloppgave.

5. (10 points) **G5:** `Game::place_new_2` **- Plasser *ett* tilfeldig 2-tall**

   Oppgaven din er å plassere *ett* 2-tall på en tilfeldig plass på spillbrettet.

   Du kan anta at det finnes minst en ledig plass, det vil si minst en plass med verdien 0.



Figur 2: Nytt spill.

6. (10 points) **G6:** `Game::flip` **- Speil spillbrettet horisontalt**

   Siden spillbrettet er et rektangel kan vi tenke på det som en matrise. I denne oppgaven skal du rokkere om på elementene i den interne representasjonen, altså rekkefølgen på elementene i `board`. Resultatet av en omrokkering skal gjøre at `at()`-funksjonen henter ut elementer i en annen rekkefølge.

   Som eksempel kan vi ta en rad fra en matrise, `[1, 2, 3, 4]`, som speilvendt blir `[4, 3, 2, 1]`. Hvis vi bestemmer at dette er første rad, vil `at(0, 0)`, `at(1, 0)` og `at(2, 0)` gi oss hhv. `1, 2, 3`, men i det speilvendte tilfellet vil de samme funksjonskallene gi oss hhv. `4, 3, 2`. Se også figur 4 for hvordan en speilvending av figur 3 ser ut i programmet.

   Tips: fra og med denne oppgaven kan det være nyttig å enkelt fylle inn debug-verdier i spillbrettet og teste dine implementasjoner av `flip()`, `transpose()`, osv. Vi har opprettet funksjonalitet som bl.a. lar deg kalle dine implementasjoner fra spillvinduet. Spillvinduet tolker enkelte tastetrykk og utfører noen handlinger som endrer spillbrettet uten å sjekke gyldigheten av spillet. Merk at du kan sette spillet i en ugyldig tilstand, men da skal du kunne trykke `'r'` for å resette spillet (dette kaller din `new_game()`-funksjon). Tabell 1 inneholder en oversikt over hvilke taster som gjør hvilken handling.

Tast	Handling
R/r	Start spillet på nytt
F/f	Speilvend spillbrettet
T/t	Transponer spillbrettet
I/i	Fyll spillbrettet med heltallsverdiene i intervallet $[1, 16]$ (figur 3)
D/d	Fyll hele spillbrettet med 2-tal 1
P/p	Fyll spillbrettet med verdi er for å teste push og merge (figur 6)
q	Lukk spillet/programmet

Tabell 1: Tastetrykk for debugging.

Figur 3: Spillbrettet fylt med tallene 1-16 (tastetrykk `'i'`).

Figur 4: Speilvending av figur 3 (tastetrykk `'f'`).

Figur 5: Transponert av figur 3 (tastetrykk `'t'`).

7. (10 points) **G7:** `Game::transpose` **- Transponer spillbrettet**

   Vi fortsetter tankegangen med at spillbrettet er en matrise. Din oppgave er å transponere spillbrettet. Å transponere en matrise betyr å speile den langs diagonalen. Det er det samme som å bytte om på kolonnene og radene. F.eks. er spillbrettet i figur 5 den transponerte av matrisen i figur 3.

   Merk at en matrise som transponeres to ganger vil være lik den opprinnelige matrisen. Det er verdt å dobbelsjekke at dette stemmer for din implementasjon.

8. (20 points) **G8:** `Game::push_right` **- Skyv alle elementer til høyre**

   Denne funksjonen skal skyve alle elementer på spillbrettet så langt til høyre som mulig.

   F.eks. vil en rad med elementene `[2 2 0 4]` etter denne operasjonen bli `[0 2 2 4]`
   og raden `[2 0 2 0]` bli `[0 0 2 2]`. Se figur 7 for resultatet av et kall til denne funksjonen med figur 6 som utgangspunkt.

   Tips: avhengig av hvordan du ønsker å løse oppgaven kan bruke en beholder som både kan `push_back()` og `push_front()`, f.eks. `std::list`.

   Tips 2: Vi har lagt inn en funksjonskall til denne funksjonen i `Game::move_right()` (G10). Det har vi gjort så du kan se resultatet av implementasjonen din når du trykker høyre piltast. Du kan også sette spillbrettet til figur 6 med tastetrykk `'p'`

9. (20 points) **G9:** `Game::merge_right` **- Slå sammen like fliser**

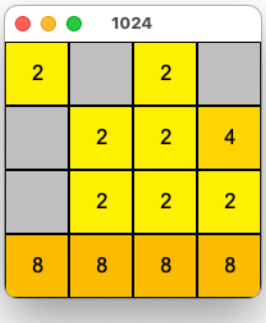
   Denne funksjonen skal slå sammen to og to elementer med lik verdi som står inntil hverandre på samme rad. Hvis det er tre elementer på samme rad med lik verdi er det de to lengst til høyre som skal slås sammen.

   Flisene som slås sammen skal danne en flisene med den totale verdien av flisene som ble slått sammen. Du kan anta at alle fliser allerede er skjøvet helt til høyre før denne funksjonen kalles (`Game::push_right` kalles før denne funksjonen) så det er ingen åpenrom mellom to brikker som har høyere verdi enn 0.

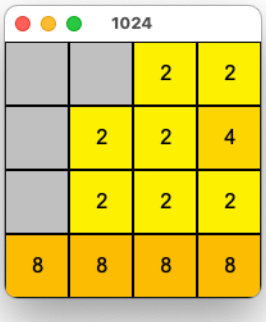
   F.eks. vil en rad med elementene `[0 2 2 4]` etter denne operasjonen bli `[0 0 4 4]`,
   raden `[0 2 2 2]` bli `[0 2 0 4]`
   og raden `[2 2 2 2]` bli `[0 4 0 4]`.

   Tips: du kan trykke `'d'` for å fylle spillbrettet med kun 2-tall.

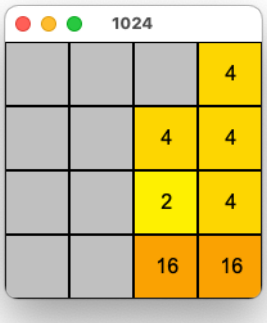
10. (10 points) **G10:** `Game::move_right` **- Spilleren flytter til høyre**



Figur 6: Spillbrettet fylt med testverdier for `push` og `merge` (tastetrykk `'p'`).



Figur 7: Et kall til funksjonen `push()` med figur 6 som utgangspunkt.



Figur 8: Forventet oppførsel av `move_right()` med figur 6 som utgangspunkt.

Denne funksjonen skal gjennomføre spillerens trekk til høyre - altså når spilleren trykker høyre piltast. La oss si at et spillebrett inneholder raden `[8 4 4 2]`, etter at denne funksjonen har gjort jobben sin skal det nye innholdet i raden være `[0 8 8 2]`. Se figur 8 for eksempel på korrekt oppførsel etter at det er gjennomført *ett* trekk med figur 6 som utgangspunkt.

Algoritmen du kan bruke for å lykkes med dette er som følger:

1. Flytt alt så langt til høyre som mulig.
2. Slå sammen fliser.
3. Flytt alt så langt til høyre som mulig.

Når du har gjort denne oppgaven skal du kunne trykke høyre piltast på tastaturet for å flytte alle brikkene til høyre i spillbrettet. Ettersom du implementerer flere retninger kan du bruke de andre piltastene til å flytte de respektive retningene.

11. (10 points) **G11:** `Game::move_down` **- Spilleren flytter nedover**

    Denne funksjonen skal gjennomføre spillerens trekk nedover. I innledningen nevnte vi at alle forflytninger er like, men i forskjellige retninger. Matriseoperasjonene, transponering og flip/speiling, du har implementert så langt skal være nok til at du kan implementere G11, G12 og G13.

    For å gjennomføre forflytninger i andre retninger skal du gjenbruke `move_right()`, men først må spillbrettet transformeres slik at *nedover* blir *høyre*. Når en forflytning er gjennomført må spillbrettet transformeres tilbake igjen.

12. (10 points) **G12:** `Game::move_left` **- Spilleren flytter til venstre**

    Som G11, men spilleren ønsker å flytte til venstre.

13. (10 points) **G13:** `Game::move_up` **- Spilleren flytter oppover**

    Som G11 og G12, men spilleren ønsker å flytte oppover.

14. (10 points) **G14:** `Game::free_spots` **- Er det noen ledige plasser?**

    Funksjonen skal returnere `true` hvis det er mulig å plassere en ny flis på brettet, med andre ord om det finnes en flis som har verdien 0. Hvis ikke skal funksjonen returnere `false`.

15. (10 points) **G15:** `Game::tick` **- Fullføring av trekk**

    Denne funksjonen kalles etter at en spiller har forsøkt å gjennomføre et trekk. Din oppgave er å sjekke om trekket spilleren forsøkte seg på flyttet noen brikker og om det fortsatt er ledige plasser igjen på brettet etter trekket. Hvis det er tilfellet skal det plasseres en ny flis med verdien 2 på brettet.

    Hvis et trekk flyttet på en brikke vil det gjenspeiles i medlemsvariabelen `bool moved`. Den er `true` hvis et trekk førte til at spillbrettet endret seg og `false` hvis spillbrettet er uendret etter at spilleren har trykket en av piltastene.

16. (10 points) **G16:** `Game::win` **- Har spilleren vunnet?**

    Hvis spillet er vunnet skal funksjonen returnere `true`, eller false hvis ikke.

    Spillet er vunnet hvis en flis på spillbrettet holder verdien vi har lagret i medlemsvariabelen `win_value`, 1024.

17. (20 points) **G17:** `Game::legal_moves` **- Er det noen gyldige trekk igjen?**

    Denne funksjonen skal sjekke om det er gyldige trekk igjen i de tilfellene det kun er fliser på brettet som har verdi over 0.

    Hvis det er mulig å gjennomføre et gyldig trekk skal funksjonen returnere `true`, ellers `false`.

    Denne funksjonen kalles etter et trekk og kun hvis vi vet at det ikke finnes noen ledige plasser på brettet. Det betyr at spillbrettet alltid vil være fylt av verdier over 0 når denne funksjonen kalles. En mulig algoritme er å finne ut av om det gjenstår gyldige trekk er å sjekke alle kolonner og rader for om det er mulig å slå sammen tilstøtende fliser. Hvis det er mulig, så finnes det gyldige trekk.