Part III: Spillet 1024

Maksimal skår for 3 er 200 poeng.

Spelet 1024 vart veldig populært i fleire utgåver og klonar i år 2014. Det går ut på at ein har eit spillbrett med 4x4 ruter, der to av rutene til å byrja med har verdien 2. Oppgåva til spelaren er å få ei av rutene i brettet til å innehalda verdien 1024. Ein spelar kan utføra fire forskjellige trekk, skyva alle brikker oppover, til høgre, nedover eller til venstre. Når brikkene blir skovne til ei av sidene vil to like brikker som ligg langs aksen det blir flytta blir lagt saman og bli ei ny brikke med summen av dei opphavlege. Viss du ikkje er kjent med spelet kan du prøva ei utgåve av det på https://poweroftwo.nemoidstudio.com/1024.

Vi har allereie implementert ein grafisk representasjon, men logikken til spelet manglar. I denne eksamensoppgåva skal du implementera stort sett all logikk for spelet over fleire deloppgåver.

Korleis besvare del 3?

Alle oppgåva i del 3 er satt opp slik at dei skal besvarast i fila `Game.cpp`. Kvar oppgåve har ei tilhørande unik kode for å gjere det lettare å finne fram til kor i fila du ska skrive svaret. Koden er på formatet `<teikn><siffer> (TS)`, eksempelvis `G1` og `G2`. I `Game.cpp` vil du for kvar oppgåve finne to kommentarar som definerer høvevis begynnelsa og slutten av koden du skal føre inn. Kommentarane er på formatet:

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

Det er veldig viktig at alle svara dine er skrivne mellom slike kommentar-par, for å støtta sensurmekanikken vår. Viss det allereie er skrive nokon kode mellom BEGIN- og END-kommentarane i filene du har fått utdelt, så kan, og ofte bør, du erstatta den koden med din eigen implementasjon.

Til dømes, for oppgåve G1 ser du følgjande kode i utdelte `Game.cpp`

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

Etter at du har implementert løysinga di, bør du enda opp med følgjande i staden for

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

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

    // END: G1
}
```

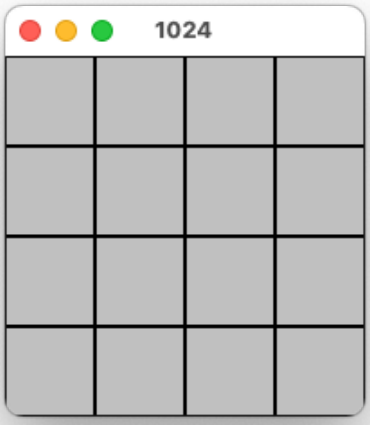
Merk at BEGIN- og END-kommentarane **IKKJE skal fjernast**.

Til slutt, viss du synest nokon av oppgåvene er uklare, si korleis du tolkar dei og skriv korleis du antek at oppgåva skal løysast som kommentarar i den koden du sender inn.

I `zip`-fila finner du m.a. `Game.h` og `Game.cpp`. Det er kun `Game.cpp` som ska redigerast for å komme i mål med oppgåvene. `Game.h` inneheld klassedefinisjonen som tar for seg logikken til spelet og er saman med `Game.cpp` filene som er viktige i denne eksamensoppgåva. `Gamewindow.h`, `Tile.h`, `settings.h` og `utilities.h` inneheld diverse deklarasjonar og definisjonar som blir brukt til den grafiske representasjonen - det er ikkje nødvendig å setja seg inn i eller forstå koden som er knytt til grafikk i denne eksamenens del 3.

Headerfila inneheld klassedefinisjoner og konstantar du bør ta ei titt på før du startar å svare på spørsmåla i denne delen.

Før du startar må du sjekke at den (umodifiserte) utdelte koden køyrar uten problem. Du skal se det samme vindauget som i figur 1. Når du har sjekka at alt fungerer som det ska er du klar til å starte programmering av svara dine.



Figur 1: Utlevert kode utan endringar.

`Game` - Spillogikk (200 poeng)

Spillets logikk skal implementerast i klassen `Game`. Han skal berre innehalda logikken til spelet og har ansvar for å korrekt flytta brikkene på brettet når spelaren vel å røra dem opp , ned, til høgre eller venstre, og dessutan starta spelet på nytt.

Spillbrettet er representert av 1D-vectoren `vector<int> board`, som inneheld board_size $*$ board_size tal heiltal. Du vil implementera ein 2D-indeksering av denne vectoren for å gjera det lettare å implementera logikken, men også tillata brukarar av klassen å indeksera eit rutenett.

Spelaren kan velja å skyva alle brikker til høgre, venstre, opp eller ned. Sidan alle operasjonar i utgangspunktet er like, men blir gjort i forskjellige retningar skal vi først implementera skyving av alle brikker til høgre og gjenbruka den metoden for å skyva brikkene i dei andre retningane. Som med alle andre program er det mogleg å implementera dette spelet på mange forskjellige måtar og i denne oppgåva skal vi m.a. bruka matriseoperasjonar til å manipulera spillbrettet så vi kan gjenbruka høyreskyvingen. Meir om dette når det er relevant i kvar enkelt oppgåve.

Vi har overlasta `operator<<(ostream&, const Game&)`, som du kan bruka for å skriva ut spillbrettet til terminalen når du måtte ønska. Merk at funksjonen ikkje vil fungera før etter oppgåve `G4` sidan funksjonane som hentar verdiane frå spillbrettet manglar implementasjonsdetaljer. Du får òg utdelt diverse funksjonar som kan brukast til å debugge logikken undervegs, sjå oppgåve `G6` og tabell 1 for meir informasjon.

Oppgavene må ikkje gjerast i ei bestemd rekkefølgje. T.d. kan funksjonen du implementerer i av deloppgåve G5 gjenbrukast i tidlegare oppgåver viss du ønsker det. Du står fritt til å bruka metodar definert kvar helst i del 3 andre stader i del 3.

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

   Funksjonen skal returnera ein verdi som tilsvarer 1D-indeksen berekna frå 2D-koordinatene til elementet med koordinatane (x, y) i spillbrettet. T.d. skal $(0, 0)$ gi 0 og $(3, 2)$ gi 11 når spillbrettet er $4x4$ stort. Bruk gjerne medlemsvariabelen `board_size`, som inneheld storleiken på spillbrettet langs ein akse, for eksamensoppgåva er det 4 (spillbrettet er 4x4).

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

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

   Funksjonene skal returnera verdien som er i spillbrettets posisjonen (x, y). Merk at det er *to* medlemsfunksjonar som skal implementerast (G2 og G3), `int Game::at(int x, int y) const` og `int& Game::at(int x, int y)`. Brikkene til spelet er lagra i medlemsvariabelen `board`.

   Viss posisjonen i spillbrettet ikkje eksisterer skal funksjonen kasta eit passande unntak av typen `std::out_of_range`.

   Vi implementerer to funksjonar her for å ha moglegheit til å henta både referansar til verdiar i spillbrettet og kopiar av verdiane. Det er formålstenleg sidan vi ønsker begge funksjonalitetene, 1) for å gjera det lettare å overskrive verdiar i spillbrettet og 2) for å lesa verdiar utan moglegheit til å redigera spillbrettet (t.d. ein spelar som ser spillbrettet skal ikkje ha moglegheit til å manipulera brikkene, det ville vore ein enkel måte å juksa i spelet på).

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

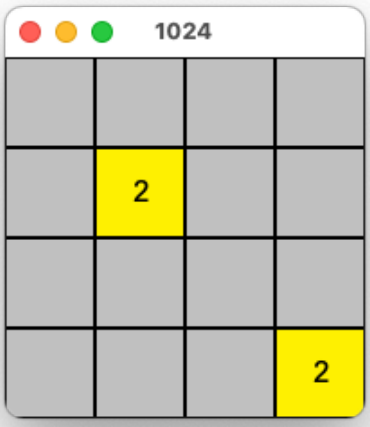
   Medlemsfunksjonen skal gjenstarte spelet så ein spelar kan byrja med blanke ark. Spillbrettet skal berre bestå av 0-verdiar, med unntak av to fliser som er tilfeldig plassert på spillbrettet. Sjå figur 2 for eit døme på korleis spillbrettet kan sjå ut etter denne funksjonen.

   Merk at du skal bruka medlemsfunksjonen `place_new_2` til å plassera 2-tala - sjå neste deloppgåve.

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

   Oppgåva di er å plassera *eitt* 2-tal på ein tilfeldig plass på spillbrettet.

   Du kan anta at det finst minst ein ledig plass, det vil seia minst ein plass med verdien 0.



Figur 2: Nytt spel.

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

   Sidan spillbrettet er eit rektangel kan vi tenka på det som ein matrise. I denne oppgåva skal du rokkere om på elementa i den interne representasjonen, altså rekkefølgja på elementa i `board`. Resultatet av ein omrokkering skal gjera at `at()`-funksjonen hentar ut element i ei anna rekkefølgje.

   Som døme kan vi ta ei rad frå ein matrise, `[1, 2, 3, 4]`, som spegelvendt blir `[4, 3, 2, 1]`. Viss 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 spegelvende tilfellet vil dei same funksjonskalla gi oss hhv. `4, 3, 2`. Sjå òg figur 4 for korleis ei spegelvending av figur 3 ser ut i programmet.

   Tips: frå og med denne oppgåva kan det vera nyttig å enkelt fylla inn debug-verdier i spillbrettet og testa dine implementasjoner av `flip()`, `transpose()`, osb. Vi har oppretta funksjonalitet som m.a. lèt deg kalla dine implementasjoner frå spillvinduet. Spillvinduet tolkar enkelte tastetrykk og utfører nokre handlingar som endrar spillbrettet utan å sjekka gyldigheten av spelet. Merk at du kan setja spelet i ein ugyldig tilstand, men då skal du kunna trykka `'r'` for å resette spelet (dette kallar din `new_game()`-funksjon). Tabell 1 inneheld ei oversyn over kva tastar som gjer kva handling.

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

Tabell 1: Tastetrykk for debugging.

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

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

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

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

   Vi fortset tankegangen med at spillbrettet er ein matrise. Oppgåva di er å transponera spillbrettet. Å transponera ein matrise betyr å spegla han langs diagonalen. Det er det same som å byta om på kolonnane og radene. T.d. er spillbrettet i figur 5 han transponerte av matrisen i figur 3.

   Merk at ein matrise som blir transponert to gangar vil vera lik den opphavlege matrisen. Det er verd å dobbelsjekke at dette stemmer for din implementasjon.

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

   Denne funksjonen skal skyva alle element på spillbrettet så langt til høgre som mogleg.

   T.d. vil ei rad med elementa `[2 2 0 4]` etter denne operasjonen bli `[0 2 2 4]`
   og rada `[2 0 2 0]` bli `[0 0 2 2]`. Sjå figur 7 for resultatet av eit kall til denne funksjonen med figur 6 som utgangspunkt.

   Tips: avhengig av korleis du ønsker å løysa oppgåva kan bruka ein behaldar som både kan `push_back()` *og* `push_front()`*, t.d.* `std::list`*.*

   Tips 2: Vi har lagt inn ein funksjonskall til denne funksjonen i `Game::move_right()` *(G10). Det har vi gjort så du kan sjå resultatet av implementasjonen din når du trykker høgre piltast. Du kan òg setja spillbrettet til figur 6 med tastetrykk* `'p'`

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

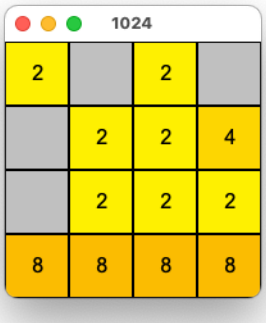
   Denne funksjonen skal slå saman to og to element med lik verdi som står inntil kvarandre på same rad. Viss det er tre element på same rad med lik verdi er det dei to lengst til høgre som skal slåast saman.

   Flisene som blir slåtte saman skal danna ein flisene med den totale verdien av flisene som vart slått saman. Du kan anta at alle fliser allereie er skove heilt til høgre før denne funksjonen kallast (`Game::push_right` kallast før denne funksjonen) så det er ingen åpenrom mellom to brikker som har høgare verdi enn 0.

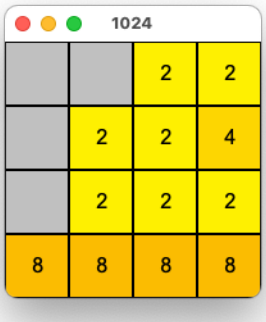
   T.d. vil ei rad med elementa `[0 2 2 4]` etter denne operasjonen bli `[0 0 4 4]`,
   rada `[0 2 2 2]` bli `[0 2 0 4]`
   og rada `[2 2 2 2]` bli `[0 4 0 4]`.

   Tips: du kan trykka `'d'` *for å fylla spillbrettet med berre 2-tal.*

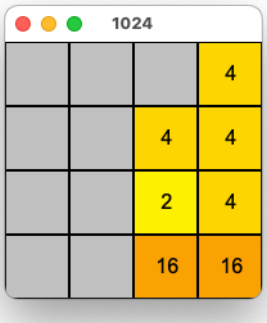
10. (10 points) **G10:** `Game::move_right` **- Spelaren flyttar til høgre**



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



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



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

Denne funksjonen skal gjennomføra trekken til spelaren til høgre - altså når spelaren trykker høgre piltast. La oss seia at eit spelebrett inneheld rada `[8 4 4 2]`, etter at denne funksjonen har gjort jobben sin skal det nye innhaldet i rada vera `[0 8 8 2]`. Sjå figur 8 til dømes på korrekt åtferd etter at det er gjennomført *eitt* trekk med figur 6 som utgangspunkt.

Algoritmen du kan bruka for å lykkast med dette er som følgjer:

1. Flytt alt så langt til høgre som mogleg.
2. Slå saman fliser.
3. Flytt alt så langt til høgre som mogleg.

Når du har gjort denne oppgåva skal du kunna trykka høgre piltast på tastaturet for å flytta alle brikkene til høgre i spillbrettet. Sidan du implementerer fleire retningar kan du bruka dei andre piltastene til å flytta dei respektive retningane.

11. (10 points) **G11:** `Game::move_down` **- Spelaren flyttar nedover**

    Denne funksjonen skal gjennomføra trekken til spelaren nedover. I innleiinga nemnde vi at alle forflytninger er like, men i forskjellige retningar. Matriseoperasjonane, transponering og flip/speiling, du har implementert så langt skal vera nok til at du kan implementera G11, G12 og G13.

    For å gjennomføra forflytninger i andre retningar skal du gjenbruka `move_right()`, men først må spillbrettet transformerast slik at *nedover* blir *høgre*. Når ein forflytning er gjennomført må spillbrettet transformerast tilbake igjen.

12. (10 points) **G12:** `Game::move_left` **- Spelaren flyttar til venstre**

    Som G11, men spelaren ønsker å flytta til venstre.

13. (10 points) **G13:** `Game::move_up` **- Spelaren flyttar oppover**

    Som G11 og G12, men spelaren ønsker å flytta oppover.

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

    Funksjonen skal returnera `true` viss det er mogleg å plassera ei ny flis på brettet, med andre ord om det finst ei flis som har verdien 0. Viss ikkje skal funksjonen returnera `false`.

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

    Denne funksjonen kallast etter at ein spelar har freista å gjennomføra eit trekk. Oppgåva di er å sjekka om trekket spelaren freista seg på flytta nokre brikker og om det framleis er ledige plassar igjen på brettet etter trekket. Viss det er tilfellet skal det plasserast ei ny flis med verdien 2 på brettet.

    Viss eit trekk flytta på ei brikke vil det gjenspeglast i medlemsvariabelen `bool moved`. Han er `true` viss eit trekk førte til at spillbrettet endra seg og `false` viss spillbrettet er uendra etter at spelaren har trykt ein av piltastene.

16. (10 points) **G16:** `Game::win` **- Har spelaren vunne?**

    Viss spelet er vunne skal funksjonen returnera `true`, eller false viss ikkje.

    Spelet er vunne viss ei flis på spillbrettet held verdien vi har lagra i medlemsvariabelen `win_value`, 1024.

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

    Denne funksjonen skal sjekka om det er gyldige trekk igjen i dei tilfella det berre er fliser på brettet som har verdi over 0.

    Viss det er mogleg å gjennomføra eit gyldig trekk skal funksjonen returnera `true`, elles `false`.

    Denne funksjonen kallast etter eit trekk og berre viss vi veit at det ikkje finst nokon ledige plassar på brettet. Det betyr at spillbrettet alltid vil vera fylt av verdiar over 0 når denne funksjonen kallast. Ein mogleg algoritme er å finna ut av om det gjenstå gyldige trekk er å sjekka alle kolonnar og rader for om det er mogleg å slå saman tilstøtande fliser. Viss det er mogleg, så finst det gyldige trekk.