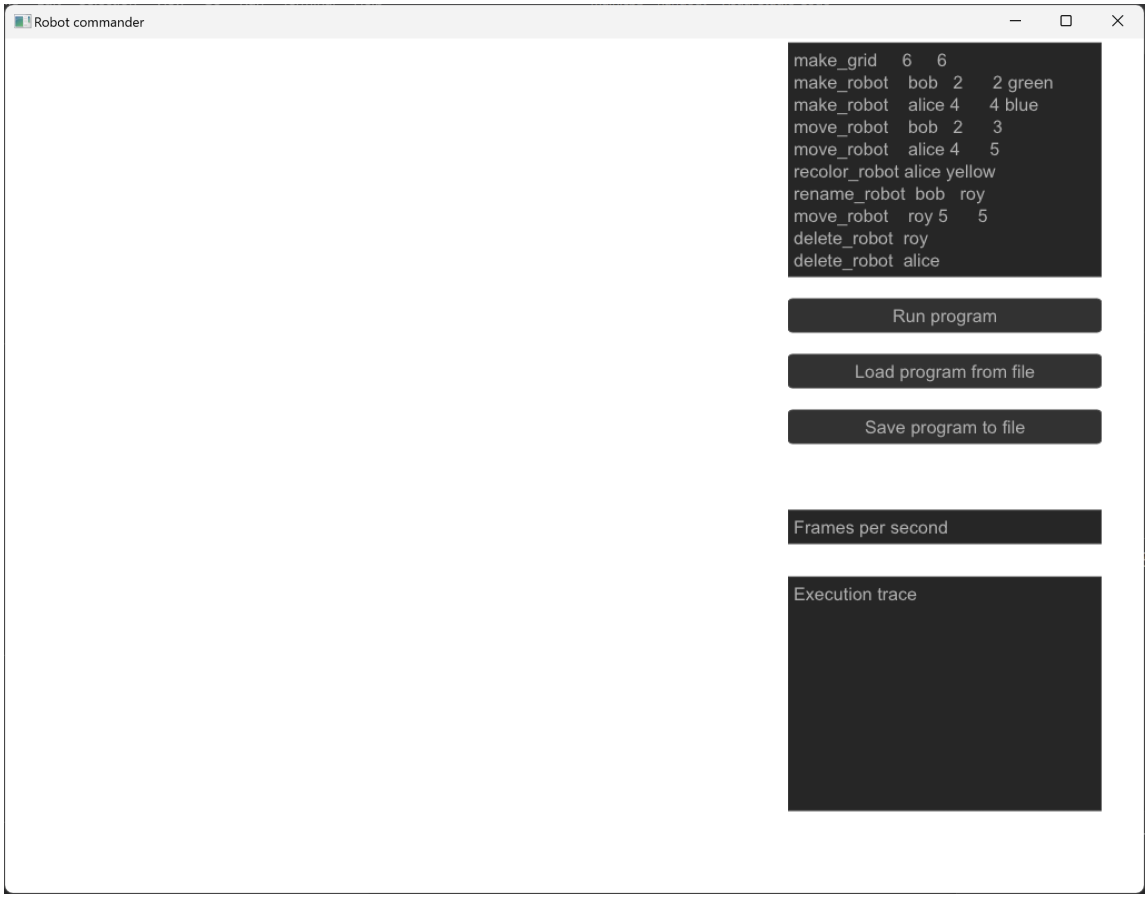
Part III: RobotCommander

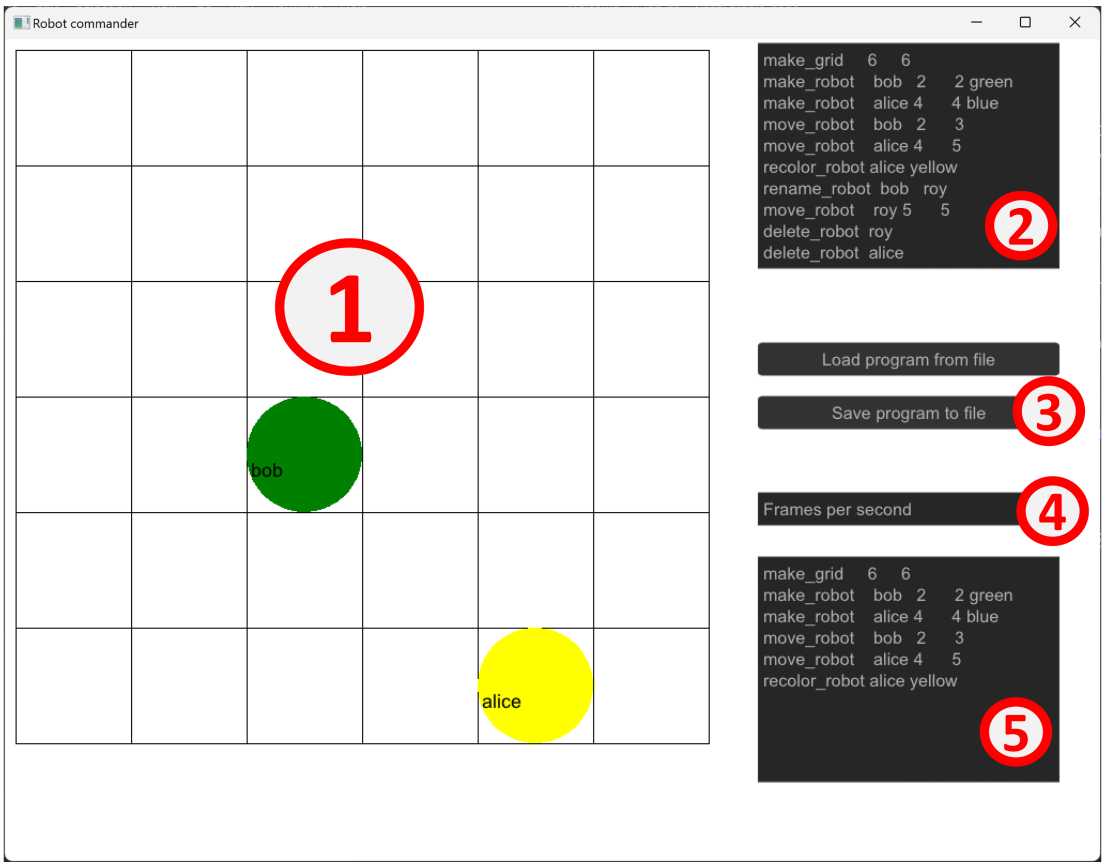
Maksimal score for del 3 er 180 poeng.

3.3 Introduksjon

RobotCommander er en applikasjon som utfører enkle kommandoer (skrevet på et bestemt, enkelt format) som flytter og endrer på "roboter" i et 2D-univers. Resultatet av kommandoene vises visuelt ved å vise robotene i et 2D rutenett med et justerbart antall rader og kolonner. I `zip`-filen handout finner du et kjørbart applikasjonsskjelett som mangler noen viktige deler av den ønskede funksjonaliteten til RobotCommander. Denne eksamensoppgaven går ut på å implementere disse manglende bitene slik at RobotCommander fungerer som ønsket. Du vil ledes gjennom denne prosessen ved hjelp av alle deloppgavene i denne delen av eksamen. Under finner du en grundig introduksjon til og forklaring av hvordan RobotCommander fungerer.



Figur 1: Skjermbilde av applikasjonen slik den ser ut når du kjører handout-koden (uten å ha endret på handout-koden). **Merk:** Programmet som ligger inne i "Control script"-vinduet som kommer når du kjører den utdelte koden er *ikke* kjørbart før du har implementert koden som kreves for hver av instruksjonene.



Figur 2: Skjermbilde av at den ferdige applikasjonen kjører et "control script". Se Avsnitt 3.4 for en forklaring av de ulike GUI elementene markert med røde tall med sirkler rundt.

3.4 Slik fungerer RobotCommander

I denne delen finner du forklaring på hvordan RobotCommander skal fungere når den er ferdig implementert. Tallene med sirkel rundt i denne delen refererer til skjermbildet dere finner i Figur 2. Merk at dette skjermbilde viser hvordan applikasjonen virker når den er ferdig, det vil si *etter* at dere har løst hele del 3 av denne eksamenen. For et skjermbilde av hvordan den ser ut når man kjører den utdelte koden, se Figur 1. Hoveddelen av applikasjonen består av rutenettet ① som representerer universet robotene eksisterer i. Robotene visualiseres basert på posisjon, farge og navn, slik dere ser i figuren. I figur 2 er roboten kalt bob grønn og står på plass $(2, 3)$, og roboten kalt alice er gul og står på plass $(4, 5)$. Kontrollskriptet ("Control script"), se nærmere forklaring i Avsnitt 3.5, er skrevet på et redigerbart tekstområde ②. Knappene ③ brukes til å kjøre kontrollskriptet (*Run program*), laste inn et eksisterende program fra en fil til "Control script"-tekstområdet (*Load program from file*), og til å lagre programmet som står skrevet i "Control script"-tekstområdet til en fil (*Save program to file*). Verdien i tekstområde ④ brukes til å kontrollere hvor raskt RobotCommander kjører kontrollskriptet. Ettersom programmet utfører kommandoene vises disse i sporingsfeltet ("Execution trace") ⑤. Dette gjør det lettere å debugge da du kan se hvilken kommando som nettopp har blitt utført for å forstå hvor feilen sannsynligvis oppsto.

3.5 Hvordan bruke RobotCommander

Robotenes egenskaper og posisjon kontrolleres ved hjelp av kontrollskript (*control scripts*). Et kontrollskript er (i denne sammenheng) en sekvens med instruksjoner som kan lage, slette, endre og flytte på roboter i rutenettet. Kontrollskriptet er strukturert slik at det er en kommando/instruksjon per linje. Disse instruksjonene gis alltid på samme format: navnet på instruksjonen, etterfulgt av null eller flere argumenter. La oss se på et eksempel på et slikt kontrollskript.

Eksempelprogram:

```
make_grid     6     6
make_robot    bob   2      2 green
make_robot    alice 4      4 blue
move_robot    bob   2      3
move_robot    alice 4      5
recolor_robot alice yellow
rename_robot  bob   roy
move_robot    roy 5      5
delete_robot  roy
delete_robot  alice
```

Først lages et nytt "univers" ved å bruke `make_grid` instruksjonen. Tallene etter `make_grid`, 6 og 6, er argumentene som sendes inn til denne instruksjonen. I dette tilfellet representerer de antall rader og kolonner, slik at vi oppretter et rutenett med 6 rader og 6 kolonner. De to påfølgende `make_robot` instruksjonene lager robotene bob og alice, med startkoordinater, $(2, 2)$ for bob og $(4, 4)$ for alice, og fargene grønn (bob) og blå (alice). `move_robot` instruksjonene flytter deretter de to robotene til nye posisjoner i rutenettet, henholdsvis $(2, 3)$ for bob og $(4, 5)$ for alice. Merk at rutenettet er nullindeksert slik at posisjon $(0, 0)$ er øverste rute til venstre og $(5, 5)$ er nederste rute til høyre i dette rutenettet. `recolor_robot` kommandoen endrer fargen til alice til gul. Deretter endrer bob navn til roy ved hjelp av `rename_robot` instruksjonen. Merk at det ikke finnes noen robot kalt bob etter denne endringen, så enhver referanse til dette navnet vil derfor være ugyldig. For å flytte roboten som før het bob må man derfor bruke det nye navnet roy i stedet. Avslutningsvis slettes de to robotene roy og alice ved å bruke `delete_robot` instruksjonen. Alternativt kunne man brukt `clear_robots` (denne tar ikke inn noen argumenter) for å slette alle robotene i rutenettet med en enkelt kommando.

Merk at instruksjonene vil feile dersom argumentene er ugyldige. Eksempler på ugyldige argumenter er blant annet:

- koordinater utenfor rutenettets grenser
- ikke-eksisterende farger
- bruk av ugyldige robotnavn, f.eks. navn som ikke er i bruk

Vi vil se nærmere på disse begrensningene etterpå når du skal implementere funksjoner for å sjekke argumentenes gyldighet.

Oversikt over instruksjonene/kommandoene som skal støttes i RobotCommander finner du i Tabell 1. Denne kan være et nyttig oppslagsverk i de senere deloppgavene.

Vi har lagt ved en kort video (`robotcommander_video.mp4`, uten lyd) i den utdelte zip-filen som viser hvordan en kjøring av kontrollskriptet som ble vist og forklart i avsnitt 3.5 i ferdig implementert RobotCommander ser ut.

Hvordan besvare del 3?

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

Tabell 1: Oversikt over instruksjonene/kommandoene som kan brukes på robotene.

Instruksjon/kommando	Beskrivelse
`make_grid <rows> <cols>`	Lager et nytt rutenett med et bestemt antall rader og kolonner. Denne kommandoen rydder opp i alle eksisterende tilstander og sletter alle roboter.
`make_robot <name> <x_pos> <y_pos> <color>`	Lager en robot kalt `name` på rutenett-posisjon (`x_pos`, `y_pos`) med farge `color`
`clear_robots`	Fjerner alle roboter fra rutenettet.
`move_robot <name> <x_pos> <y_pos>`	Flytter roboten kalt `name` til rutenett-posisjonen (`x_pos`, `y_pos`)
`recolor_robot <name> <color>`	Endrer fargen til roboten kalt `name` til `color`
`rename_robot <name> <new_name>`	Endrer navnet til roboten kalt `name`. Endrer fra `name` til `new_name`
`delete_robot <name>`	Sletter roboten kalt `name`

fått utdelt, så kan, og ofte bør, du erstatte den koden med din egen implementasjon med mindre annet er spesifisert i oppgavebeskrivelsen. All kode som står *utenfor* BEGIN- og END-kommentarene SKAL dere la stå.

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

```
void RobotGrid::draw_grid_lines()
{
  // BEGIN: G1
  //
  // Write your answer to assignment G1 here, between the // BEGIN: G1
  // and // END: G1 comments. You should remove any code that is
  // already there and replace it with your own.

  // END: G1
}
```

Etter at du har implementert din løsning, bør du ende opp med følgende i stedet:

```
void RobotGrid::draw_grid_lines()
{
  // BEGIN: G1
  //
  // Write your answer to assignment G1 here, between the // BEGIN: G1
  // and // END: G1 comments. You should remove any code that is
  // already there and replace it with your own.

  /* Din kode her */

  // END: G1
}
```

Tabell 2: Oversikt over koblingen mellom oppgave-bokstav og filnavn.

Bokstav	Fil
A	`application.cpp`
S	`interpreter.cpp`
G	`robot_grid.cpp`

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.

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.

3.5.1 Hvor i koden finner du oppgavene?

Hver oppgave har en unik kode bestående av an bokstav etterfulgt av et tall. Bokstaven indikerer hvilken fil oppgaven skal løses i. Se Tabell 2 for en oversikt over bokstaver og filnavn. Eksempelvis finner du oppgave A1 i filen `application.cpp`

3.6 Oppgavene:

Lage og modifisere roboter (90 poeng)

I denne første delen av oppgavene skal du implementere kode for å lage, modifisere og tegne robotene. Men først har vi en liten forklaring av visualiseringsmodellen til RobotCommander og hvordan robotene representeres.

En robot er representert av structen `Robot` som er definert under. Denne er definert i filen `robot.h` i den utdelte koden. Alle roboter i universet er instanser av denne structen.

```
1 struct Robot {
2   Robot(string name, Point pos, Color color);
3
4   string name;
5   Point pos;
6   Color color;
7 };
```

Koden for å manipulere og tegne robotene ligger i filen `robot_grid.cpp` og i klassen `RobotGrid`. Denne klassen har et map `robots` definert som

```
1 map<string, unique_ptr<Robot>> robots;
```

som kobler navnet til robotene med instanser av structen `Robot`.

Funksjonene `RobotGrid::draw_grid_lines()` og `RobotGrid::draw_robots()`, definert i `robot_grid.cpp`, tegner deler av den grafiske representasjonen til universet, nærmere bestemt linjene i rutenettet og robotene. Se Figur 2 ① for et eksempel på hvordan dette vil kunne se ut. Disse funksjonene tegner hele universet på nytt hver gang de kalles. Du slipper derfor å ta hensyn til visualiseringens tilstand. Du skal implementere disse funksjonene i oppgave G1 og G4. Denne måten å designe programmet på gjør at alle endringer i universet vises i den grafiske representasjonen umiddelbart, uten at man eksplisitt må kalle f.eks. en GUI "update"-funksjon.

1. (10 points) **G1: Tegn linjene i rutenettet**

   Implementer koden for å tegne linjene i rutenettet, mao. `RobotGrid::draw_grid_lines()`-funksjonen. For å løse denne oppgaven bør du bruke `window.draw_line()` funksjonen for å tegne linjene og følgende medlemsvariabler i `RobotGrid`-klassen for å bestemme rutenettets utseende:

   - `x_pos` og `y_pos`: koordinatene til øvre venstre hjørne i rutenettet
   - `w_size` ogh_size: bredden og høyden til rutenettet
   - `cell_width` og `cell_height`: bredden og høyden til hver celle i rutenettet.

2. (10 points) **G2: Finn midten av en celle i rutenettet**

   Skriv funksjonen `RobotGrid::get_grid_cell_center_coord()` som, gitt `x` og `y` koordinatene til en rutenett-celle, returnerer skjerm-koordinatene til cellens sentrum. For eksempel vil sentrum i rutenett-celle $(1, 1)$ kunne være $(150, 150)$ piksler fra øvre venstre hjørne i applikajonsvinduet (avhengig av cellenes bredde og høyde etc). Du kan regne ut dette vha. de samme medlemsvariablene som vi tipset om i oppgave G1.

   ```
      |     |
   ---+-----+---
      |  x  |
   ---+-----+---
      |     |
   ```

   For å illustrere, skissen over viser et rutenett. Funksjonen skal returnere skjermkoordinatene til en vilkårlig celle `x`. **Merk:** Du kan anta at koordinatene som sendes til funksjonen er innenfor rutenettets grenser. Vi håndterer unntak i en senere oppgave.

3. (10 points) **G3: Lag en robot**

   Implementer koden for å lage en ny robot, `RobotGrid::make_robot()`-funksjonen. Du gjør dette ved å lage en instans av `Robot`-structen (definert i `robot.h`) og legge til denne i `robots`-mapet i `RobotGrid`-klassen. Du trenger ikke å sjekke om posisjonen og navnet er gyldig, dette håndteres i en senere oppgave. Husk at `robots` er et map fra `string` til `unique_ptr<Robot>`.

   Du må gjøre oppgave G4 (tegne robot) og oppdatere deler av koden beskrevet i oppgave S2 for å teste denne funksjonen. Merk at robotene ikke får korrekt farge før oppgave S1 har blitt løst.

4. (10 points) **G4: Tegning av robotene**

   Skriv kode for å tegne robotene i rutenettet, `RobotGrid::draw_robots()`-funksjonen. En robot representeres av en sirkel som tegnes innenfor grensene til rutenettcellen roboten er plassert i. Sirkelen er fylt med robotens farge, og robotens navn er skrevet oppå sirkelen med startpunkt midt på venstre side av cellen. Du trenger ikke sjekke om posisjonen og navnet er gyldig, dette håndteres i en senere oppgave.

   For å vite hvor du skal plassere robot-sirklene kan det være lurt å bruke `get_grid_cell_center_coord()`-funksjonen du nettopp har implementert. Funksjonen `get_grid_cell_edge_coord()`, som er en del av den utdelte koden, kan også være nyttig i denne oppgaven.

   Du må oppdatere deler av koden beskrevet i oppgave S2 for å kunne teste denne funksjonen.

5. (10 points) **G5: Slett en robot**

   Implementer koden for å slette en robot fra rutenettet, `RobotGrid::delete_robot()`-funksjonen. Husk at alle robotene i rutenettet er listet i `robots` map-et.

   Du må oppdatere deler av koden beskrevet i oppgave S2 for å kunne teste denne funksjonen.

6. (10 points) **G6: Flytt en robot**

   Implementer koden for å flytte på en robot, altså funksjon `RobotGrid::move_robot()`. Gjør dette ved å endre på den korresponderende instansen av `Robot` i `robots`- map-et.

   Du må oppdatere deler av koden beskrevet i oppgave S2 for å kunne teste denne funksjonen.

7. (10 points) **G7: Endre fargen til en robot**

   Implementer kode for å endre fargen på en robot, mao. funksjon `RobotGrid::recolor_robot()`.

   Du må løse oppgave S1 og oppdatere deler av koden beskrevet i oppgave S2 for å kunne teste denne funksjonen.

8. (10 points) **G8: Tøm rutenettet**

   Implementer koden for å fjerne alle robotene fra rutenettet, altså funksjon `RobotGrid::clear_robot()`.

   Du må oppdatere deler av koden beskrevet i oppgave S2 for å kunne teste denne funksjonen.

9. (10 points) **G9: Gi roboten nytt navn**

   Implementer koden for å gi en robot nytt navn, i.e. `RobotGrid::rename_robot()`-funksjonen. Husk å endre robotens nøkkel i `robots` map-et!

   Du må oppdatere deler av koden beskrevet i oppgave S2 for å kunne teste denne funksjonen.

Implementering av instruksjonstolkeren (20 poeng)

10. (10 points) **S1: Fra string til Color**

    Denne funksjonen tar inn navnet på en farge som en `string` og returnerer Color sin versjon av denne fargen. Denne funksjonen er dermed nødvendig for å kunne konvertere fargenavnene som gis som tekststreng-argumenter til de ulike instruksjonene til datatypen Color som koden trenger for å kunne fargelegge robotene i rett farge. I den utdelte koden finner du map-et `color_map` i filen `interpreter.h`. Denne inneholder alle gyldige koblinger mellom fargenavn og farger.

    Din oppgave er å skrive funksjonen `Interpreter::get_color()` som finner Color-versjonen av fargen ved hjelp av `color_map`-map-et. Dersom fargen ikke finnes i map-et skal det kastes et unntak med en beskrivende feilmelding.

11. (10 points) **S2: Instruksjonstolkeren**

    Instruksjonstolke-funksjonen `Interpreter::execute_instruction()` leser en instruksjon (som beskrevet i Tabell 1) og kaller den korresponderende funksjonen i `RobotGrid`-klassen for å utføre den ønskede handlingen på universet.

    For å hjelpe deg i gang har vi gitt strukturen til funksjonen og implemenert `make_grid` instruksjonen. Din oppgave er å implementere hvordan alle de andre instruksjonene skal håndteres. Alle instruksjonene kan implementeres med samme metode/teknikk som den vi brukte for å implementere `make_grid`-instruksjonen. Det er opp til deg om du vil følge denne strukturen eller ikke. Dersom du er usikker på hvilken funksjon fra `RobotGrid`-klassen du skal kalle på kan det være lurt å se etter en funksjon som har samme navn som instruksjonen.

    Denne funksjonen, altså `Interpreter::execute_instruction()`, kalles en gang for hver eneste instruksjon i programmet. Dette innebærer at for det følgende kontrollskriptet

    ```
    make_robot ayesha 1 1 blue
    move_robot ayesha 2 4
    ```

    blir denne funksjonen kalt to ganger, en gang for hver instruksjon. `instruction`-argumentet til funksjonen er en `istringstream` som inneholder en linje fra kontrollskriptet. Med andre ord vil de to `execute_instruction`-funksjonskallene som er vist under utføres i koden når kommandoene over kjøres i GUI-vinduet.

    ```
    execute_instruction(istringstream("make_robot ayesha 1 1 blue"));
    execute_instruction(istringstream("move_robot ayesha 2 4"));
    ```

Sikre at kun lovlige instruksjoner kjøres (40 poeng)

I de neste oppgavene skal du implementere funksjoner om sjekker om en instruksjon og dens parametere er gyldige eller ikke. Universet må til en enhver tid opprettholde en rekke egenskaper, se listen under. En instruksjon er ugyldig dersom vi oppdager at universets egenskaper ikke vil bli opprettholdt dersom intruksjonen hadde blitt utført. Universets egenskaper er:

- Alle robotnavn må være unike.
- Det kan *ikke* stå flere roboter i samme rute/celle i rutenettet.
- Alle roboter må stå på en gyldig posisjon i rutenettet.
- Instruksjoner kan bare referere til/brukes på eksisterende roboter.
- Alle roboter må ha en gyldig farge.

12. (10 points) **G10: Sjekk av koordinatene**

    Denne funksjonen, `RobotGrid::check_coord_bounds()`, sjekker om koordinatene som har blitt sendt inn til funksjonen som punktet `p` er gyldige eller ikke. Gyldige koordinater er: $0 <= x < cols$ og $0 <= y < rows$. Funksjonen skal kaste et unntak med en beskrivende feilmelding dersom koordinatene er ugyldige. Dersom de er gyldige skal ikke funksjonen gjøre noe.

13. (10 points) **G11: Sjekk om robotnavnet er ledig**

    Denne funksjonen, `RobotGrid::check_name_available()`, sjekker om `name` er ledig (og dermed kan brukes av en robot). Dette er tilfellet dersom parameteren `name` *IKKE* er en nøkkel i `robots` map-et. Kast et unntak med en beskrivende feilmelding dersom navnet allerede er i bruk.

14. (10 points) **G12: Sjekk om roboten eksisterer**

    Denne funksjonen, `RobotGrid::check_name_exists()`, sjekker om en robot kalt `name` eksisterer. Dette er tilfellet dersom `name` er en nøkkel i `robots` map-et. Dersom navnet *IKKE* er i bruk skal det kastes et unntak med en beskrivende feilmelding.

15. (10 points) **G13: Sjekk om en rute i rutenettet er ledig**

    Denne funksjonen, `RobotGrid::check_coord_empty()`, sjekker om det er en robot i ruten med koordinatene gitt som punktet `p`. Parameteren `is_moving` er `true` dersom funksjonen kalles når det er instruksjonen `move_robot` som blir forsøkt utført, og `false` dersom funksjonen kalles når det er instruksjonen `make_robot` som blir forsøkt utført. `name` er navnet på roboten som blir flyttet eller laget. Skillet mellom flytting og laging er viktig da det er tillatt å flytte en robot til seg selv. Det vil si at programmet:

    ```
    make_robot tao 1 1 blue
    move_robot tao 1 1
    ```

    er gyldig, mens programmet:

    ```
    make_robot tao 1 1 blue
    make_robot fatima 1 1 green
    ```

    ikke er det.

    Funksjonen skal kaste et unntak med en beskrivende feilmelding dersom ruten/cellen allerede inneholder en annen robot. Husk på at det er lov å flytte til seg selv som beskrevet over.

Fil I/O og verdi-konvertering (30 poeng)

I den siste delen av denne eksamenen skal vi implementere de siste manglende delene av brukergrensesnittet: Laste inn fra og lagre programmer til filer, og få programmet til å utføre instruksjonene i hastigheten gitt av verdien i "Frames per second"-tekstfeltet.

16. (10 points) **A1: Laste inn et program fra en fil**

    Implementer funksjonen `Application::load_program()` for å lese innholdet i en fil gitt ved parameteren `file_name` og returner filens innhold som en string. Pass på å inkludere whitespace (linjeskift, mellomrom etc) fra filen i stringen. Dette gjør det mulig å laste inn et program fra en fil til `Control script`-vinduet i applikasjonen. Et par test-program har blitt inkludert i den utdelte koden. Du kan bruke disse til å teste funksjonen dersom du ikke har tid/lyst til å lage dine egne testfiler. Dersom man ønsker å endre hvilken fil man laster inn, eller hva filen man lagrer heter kan man endre strengen i henholdsvis `cb_btn_load_program()` og `cb_btn_save_program()`. Eventuelle feil som oppstår når filen blir forsøkt åpnet skal håndteres på en fornuftig måte.

17. (10 points) **A2: Lagre et program i en fil**

    Implementer funksjonen `Application::save_program()` for å skrive innholdet i string-parameteren `contents` til en fil med filnavnet gitt av parameteren `file_name`. Dette gjør det mulig å lagre det nåværende skriptet i `Control script`-vinduet til en fil. Eventuelle feil som oppstår når filen blir forsøkt åpnet skal håndteres på en fornuftig måte.

18. (10 points) **A3: Sjekke om en `string` er et heltall**

    Implementer funksjonen `Application::is_int()` for å sjekke om en `string` er et `int`. Funksjonen skal returnere `true` hvis inndataen kan konverteres til et heltall og `false` ellers.