

Norges teknisk–naturvitenskapelige
universitet
Institutt for datateknologi og
informatikk

TDT4102 Prosedyre- og objektorientert programmering Vår 2024

Øving 9

Frist: 2024-03-22

Mål for denne øvingen:

- Lære om operatører og interaksjon mellom klasser av ulike typer.
- Lære å implementere og bruke klasser.
- Lære å bruke enkle klasser for et enkelt grafisk brukergrensesnitt (GUI).
- Lære å bruke pekere
- Lære å bruke `unique_ptr` og `shared_ptr`

Generelle krav

- Bruk de eksakte navn og spesifikasjoner gitt i oppgaven.
- Teorioppgaver besvares med kommentarer i kildekoden slik at læringsassistenten enkelt finner svaret ved godkjenning.
- **Denne øvingen skal implementeres uten hjelp fra `std.lib_facilities.h`.**
- 70% av øvingen må godkjennes for at den skal vurderes som bestått.
- Øvingen skal godkjennes av stud.ass. på sal.
- Det anbefales å benytte en programmeringsomgivelse (IDE) slik som Visual Studio Code.

Anbefalt lesestoff:

- Kapittel 17, 19.5.4 og 19.5.5 i PPP
- **Dokumentasjon** til `AnimationWindow` -> GUI.

Del 1: NTNU-samkjøring (60 %)

Du har fått sommerjobb i det nye studentforetaket «EcoTrans» som er startet av noen miljøbevisste NTNU-studenter. Omorganiseringen til nye NTNU har skapt et behov for koordinert transport mellom byene Trondheim, Ålesund og Gjøvik. Hver uke reiser mange ansatte og studenter mellom disse byene, ofte i privatbiler med ledige seter. EcoTrans vil lage et enkelt datasystem for å stimulere til miljøvennlig samkjøring, og i denne oppgaven skal du skrive noen kodesnutter for et slikt system.

1 Car-klasse (10%)

a) Deklarer ei klasse Car.

Car skal ha et heltall `freeSeats` som privat medlemsvariabel som indikerer hvor mange ledige seter det er i bilen. Car skal også ha to `public` medlemsfunksjoner `hasFreeSeats` og `reserveFreeSeat`. `hasFreeSeats` returnerer `true` om bilen har ledige seter, og `false` ellers. `reserveFreeSeat` «reserverer» et ledig sete ved å dekrementere `freeSeats`-variabelen (du kan gå ut i fra at funksjonen bare blir kalt på om det er ledige seter).

Deklarasjoner for medlemsfunksjonene:

```
bool hasFreeSeats() const;
void reserveFreeSeat();
```

Nyttig å vite: Const correctness

Det er god praksis å markere medlemsfunksjoner som ikke endrer objektet med `const`. Dette gjør det enklere å finne feil i koden, og lar oss bruke medlemsfunksjonene selv om objektet er konstant.

<pre>class NumberClass { int number; public: NumberClass(int number) : number{number} {} // markert const int getNumber() const { return number; } // ikke markert const void setNumber(int newNumber) { number = newNumber; } }</pre>	<pre>int main () { NumberClass x{3}; int i = x.getNumber(); // OK x.setNumber(i+1); // OK const NumberClass y(4); int j = y.getNumber(); // OK // IKKE OK! // Kompileringsfeil: // kan ikke kalle en funksjon // som ikke er markert const, // på et const objekt y.setNumber(j+1); }</pre>
--	--

b) Deklarer og implementer en konstruktør som tar inn hvor mange ledige seter bilen har.

c) Implementer `hasFreeSeats()` og `reserveFreeSeat()`.

Husk at deklarasjonen skal være i `Car.h`, og definisjonen (implementasjonen) skal være i `Car.cpp`.

Nyttig å vite: `std::unique_ptr` og `std::shared_ptr`

Hva er en `std::unique_ptr`? `std::unique_ptr` er en smartpeker som eier og håndterer et annet objekt gjennom en peker, og sletter automatisk minnet som tilhører objektet når `std::unique_ptr`-instansen blir destruert. Når en `std::unique_ptr` blir opprettet kaller den automatisk `new`, og når den går ut av skop kaller den `delete`. Dette vil si at vi slipper å bruke `new` og `delete` når vi skal bruke dynamisk allokerede objekter. Merk at dette faget bare er en liten intro til det man kan gjøre med `std::unique_ptr`.

Hvordan opprette en `std::unique_ptr`? C++14-funksjonen `std::make_unique` brukes for å lage et nytt `std::unique_ptr`-objekt. Det allokerer også nødvendig minne til objektet `std::unique_ptr` håndterer.

```
#include <memory> //For unique_ptr
/* Her opprettes en unique_ptr som håndterer et Student-objekt.
Argumentene til funksjonen blir gitt direkte til Student-konstruktøren. */
std::unique_ptr<Student> s1 = std::make_unique<Student>("daso",
"daso@stud.ntnu.no");
/* Vi kan også bruke auto når vi oppretter en unique_ptr. */
auto s2 = std::make_unique<Student>("lana", "lana@stud.ntnu.no");
```

Hvordan bruke en `std::unique_ptr`? Derefereringsoperatoren (*) og piloperatoren (->) blir brukt som om det er en vanlig peker.

```
std::cout << s1->getName() << '\n';
std::cout << *s2 << '\n';
```

En `std::unique_ptr` eier objektet det peker til (bare en `std::unique_ptr` kan peke på et objekt om gangen) — den tillater ikke pekeren å bli kopiert. (Hvis vi hadde kunnet kopiere en `std::unique_ptr` og en av pekerne hadde gått ut av skop, ville objektet blitt slettet, og de gjenværende `std::unique_ptr` ville peke på minne vi ikke vet hva inneholder.) Derimot kan eierskapet overføres vha. `std::move`. Vi sier at `std::move` overfører eierskapet av objektet.

```
/* Overfører eigarskap frå s1 til s3. */
std::unique_ptr<Student> s3 = std::move(s1);
/* Verdien til s1 er nå udefinert. */
/* Kodelinjen under vil ikke kompilere, for std::unique_ptr kan ikke bli kopiert*/
// auto s3 = s1;
```

Etter denne operasjonen er `s3` en `std::unique_ptr` som eier det `Student`-objektet som `s1` tidligere eide. `s1` er derimot i en «gyldig men uspesifisert» tilstand.

Noen ganger ønsker vi å la andre bruke objektene som blir pekt på av `std::unique_ptr`-instansen uten å overføre eierskapet. Vi kan da få tak i den underliggende pekeren, vha. medlemsfunksjonen `get`.

```
void printStudent(Student* sPtr);
printStudent(s2.get()); /* s2.get() returnerer den underliggende
pekeren.*/
```

Medlemsfunksjonen `get` kan også bli brukt til å sjekke om en `std::unique_ptr` har et tilknyttet objekt, ved å sammenlikne med `nullptr`.

```
if (s1.get() != nullptr) {
std::cout << "S1 contains an object\n";
} else {
std::cout << "S1 does not contain an object\n"; // <- Dette skrives ut
}
if (s3.get() != nullptr) {
std::cout << "S3 contains an object\n"; // <- Dette skrives ut
} else {
std::cout << "S3 does not contain an object\n";
}
}
```

`std::unique_ptr` kan virke vanskelig, men ikke overkompliser det! Tenk på det som en vanlig peker, bare at den eier objektet den peker på og dermed har ansvar for å fiks minnet til objektet selv. Og siden det blir krøll hvis flere skal eie det samme objektet, kan man ikke ha mer enn én `std::unique_ptr` til et objekt, så dersom en annen `std::unique_ptr` skal peke dit må man «`std::move`» eigarskapet.

`std::shared_ptr` fungerer på mange måter likt, men objektet den peker på kan nå bli eid av flere `std::shared_ptr`s. For hver peker du da har til objektet vil en teller legge til én. Hvis pekerene går ut av skop, eller blir fjerna, vil telleren reduseres med én, og først når den siste `std::shared_ptr`-en går ut av skop, blir objektet slettet.

2 Person-klassen (20%)

a) Deklarer en klasse **Person**.

Denne skal ha de private medlemsvariablene `name` og `email`, begge av typen `string`. I tillegg skal klassen ha en `std::unique_ptr<Car>`, som er en peker til en `Car`. Legg merke til at vi ønsker å bruke en peker og ikke referanse til `Car`-objektet. Grunnen til at vi ønsker å bruke en peker er fordi en peker kan ha verdien `nullptr` og det passer fint for å representere at en person *ikke* har bil. Om vi bruker referanse istedenfor peker ville det vært vanskeligere å representere dette. Dette er godt forklart i læreboka §17.9.1, og mot slutten av det avsnittet er det oppsummert når man anbefaler pass-by-value, peker, eller referanse-parameter.

Klassen skal ha en konstruktør som setter `name`, `email` og `car` til verdier gitt av parameterlisten. For `car` bruker vi `nullptr` som et såkalt «default argument» (standard-verdi). Det betyr at konstruktøren kan brukes med bare de to første parametrene,

og da vil den tredje få denne standard-verdien. Se også nyttig-å-vite-boks om dette temaet. Deklarer en `get`-funksjon både for `name` og `email`. Deklarer også en `set`-funksjon for `email`. **Hint:** For å bruke `std::unique_ptr` må man inkludere headerfilen `<memory>`. For `std::string` må headerfilen `<string>` inkluderes.

Nyttig å vite: default arguments

For å unngå at man skal definere flere ulike funksjoner som gjør det samme, men har ulikt antall på parametere i parameterlisten, så finnes det default arguments. For eksempel kan en funksjon som alltid skal legger sammen to tall være standardisert til å inkrementere det første argumentet med en, dersom det andre argumentet ikke er oppgitt. Istedenfor å lage to funksjoner som gjør samme arbeid eller kaller på en annen, er det hensiktsmessig å kombinere de. Dette kan man også bruke med medlemsfunksjoner i klasser.

```
void Adder(int a, int b = 1);
//void Adder(int a = 1, int b); // Gir kompileringsfeil

int main() {
    Adder(42, 42); // a+b=84
    Adder(42); // a+b=43
}

void Adder(int a, int b) {
    cout << "a+b=" << a+b;
}
```

Default argument skrives i deklarasjonen, men ikke i definisjonen. Argumenter som har default-verdi må også skrives sist i parameterlisten.

b) **Implementer konstruktøren og `get`-/`set`-funksjonene fra forrige deloppgave.** Bruk initialiseringsliste i konstruktøren.

c) **Lag medlemsfunksjonen `hasAvailableSeats()`.**

Funksjonen returnerer `true` om personen eier en bil og bilen har ledige seter.

d) **Overlast operator<<, som skal skrive ut innholdet i `Person` til en ostream.**

Drøft:

- Hvorfor bør denne operatoren deklarerer med `const`-parameter? (t.d. `const Person& p`)
- Når bør vi, og når bør vi ikke (ev. kan ikke) bruke `const`-parameter?

Husk å inkluder `<iostream>`

e) **Skriv tester for `Person` i `main()`.**

Opprett flere personer, og prøv å test ulike tilfeller (t.d. har personen bil? Hva med når personen ikke har bil?).

3 Meeting-klassa (30%)

- a) Deklarer en `scoped enum`, med navn `Campus`, som innehold verdier for de ulike byene (Trondheim, Ålesund og Gjøvik). Overlast `operator<<` for `Campus`, som skal skrive campusnavnet til en `ostream`.

La deklarasjonen av `enum class Campus` ligge i `Meeting.h`.

- b) Definer klassen `Meeting`.

Klassen skal ha følgende private medlemsvariabler:

```
int day;
int startTime;
int endTime;
Campus location;
std::string subject;
const std::shared_ptr<Person> leader;
vector<std::shared_ptr<Person>> participants;
```

Implementer `get`-funksjoner for `day`, `startTime`, `endTime`, `location`, `subject`, og `leader` som en del av klassedefinisjonen. Husk å inkludere nødvendige headerfiler, for eksempel `<vector>`

- c) Lag medlemsfunksjonen `addParticipant`.

Den skal ta inn en `std::shared_ptr` til et `Person` objekt og legge det inn i `participants`.

- d) Lag en konstruktør for `Meeting`-klassen som tar inn `day`, `startTime`, `endTime`, `location`, `subject`, og `leader`.

Husk at møtelederen også er en deltaker.

- e) **Teori:** Når vi oppretter et `Meeting`-objekt, hvordan blir det allokeret minnet ryddet opp når vi sletter objektet eller stopper programmet?

- f) Lag funksjonen `getParticipantList()`.

Dette skal være en medlemsfunksjon i `Meeting`. Funksjonen har ingen parametere og returnerer en `std::vector<std::string>` med navn på deltakerane.

- g) Overlast `operator<<` for `Meeting`.

Denne operatoren skal IKKE være en `friend` av `Meeting`. Du står fritt til å velge format selv, men du skal skrive ut `subject`, `location`, `startTime`, `endTime`, og navnet på møtelederen. I tillegg skal den skrive ut en liste med navnet på alle deltakerene.

Test funksjonen din fra `main()`.

- h) Skriv funksjonen `findPotentialCoDriving`.

Dette skal være en medlemsfunksjon i `Meeting`. Funksjonen skal ta inn et annet møte, og skal returnere en vektor med `Person`-pekere. Vektoren skal bestå av alle personer fra det andre møtet som:

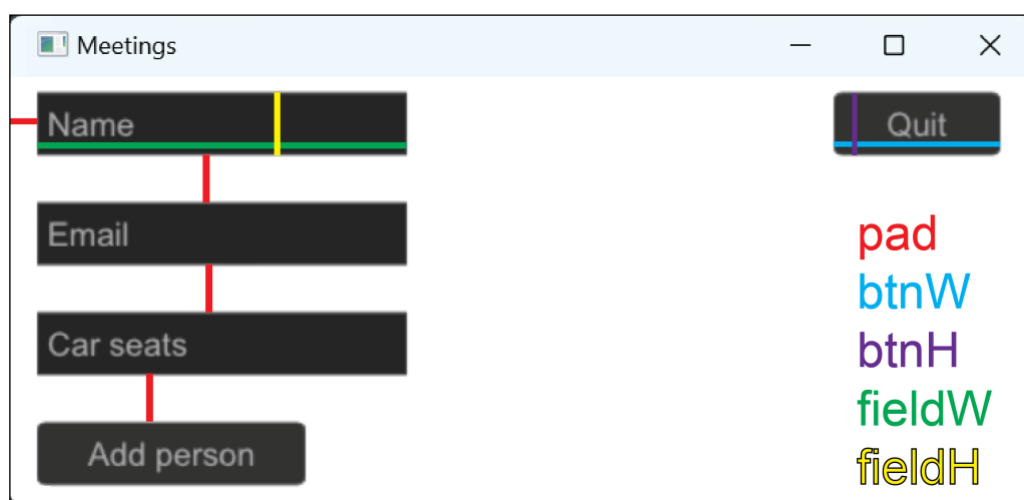
- har ledige plasser i bilen sin og
- der det andre møtet
 - er på samme sted som `this`-møtet,
 - er på samme dag som `this`-møtet,
 - har start-tid som er mindre enn en time forskjellig fra start-tida til `this`-møtet, og
 - har slutt-tid som er mindre enn en time forskjellig fra slutt-tida til `this`-møtet.

*Hint: Funksjonen vil ha følgende returtype: `std::vector<std::shared_ptr<Person>>`.
Husk `const correctness`.*

DEL 2: GUI for samkjøring og møteplanlegging (40%)

For å gjøre programmet mer brukervennlig vil dere lage et grafisk brukergrensesnitt (GUI) der passasjerer kan melde seg inn. Den skal ha to tekstfelt, et for å skrive inn passasjerens navn og et for e-post. Det skal også være to knapper: en for å legge til personen og den andre for å avslutte programmet.

Før du går løs på kodingen til et GUI, er det veldig lurt å danne seg en skisse for hvordan vinduet skal se ut. Det er også lurt å navngi alle ulike avstander i vinduet. Dette er delvis fordi man ikke trenger å sjonglere alle de ulike verdiene i hodet, men mest fordi man da kan endre på alle felles avstander på et sted. Under kommer en mulig skisse av GUI-et der hver farge er koblet til en variabel. Det er ikke et krav at du følger denne skissen, men alle elementer skal være med til slutt.



Alle variabler og funksjoner i resten av oppgavene skal være medlem av vinduet-klassen vi lager.

4 Oppretting av GUI (20%)

I denne oppgaven skal vi bruke `AnimationWindow` til å lage et vindu, der vi overlater programstyringa til selve vinduet. I praksis betyr dette at vi må lage vår egen vindu-klasse som arver fra `AnimationWindow`, og at klassen skal styre programmet.

Videre skal vi bruke widgetsene `TDT4102::TextInput` og `TDT4102::Button`, og man må inkludere `widgets/TextInput.h` og `widgets/Button.h` for å bruke de.

- a) Lag en ny klasse `MeetingWindow`, som arver public fra `AnimationWindow`, og konstruktøren `MeetingWindow(int x, int y, int w, int h, const string& title)`.

Plasser klassedeklarasjonen i `MeetingWindow.h`. Siden `AnimationWindow` ikke har en default-konstruktør, så må du kalle på `AnimationWindow`-konstruktøren i initialiseringslista, der argumentene skal være de du tok inn i `MeetingWindow`-konstruktøren. La konstruktørkroppen stå tom inntil videre.

b) Lag et `MeetingWindow`-objekt i `main()`.

Lag et objekt av typen `MeetingWindow` i `main()`. Prøvekjør koden, du skal få opp et blankt vindu. *Tips: bruk `wait_for_close()` funksjonen for at vinduet skal være oppe fram til brukeren lukker det.*

- c) Før du begynner på å legge inn elementer, er det lurt sette inn noen heltall i klassen som definerer oppsettet i vinduet. Se skissen over for et forslag. Siden disse ikke skal endres og er definerte før kompileringen, er det lurt å deklarere de `static constexpr int` og gi dem verdier direkte i `.h` fila.

Nyttig å vite: `static` og `static constexpr`

En statisk medlemsvariabel er en variabel som er felles for alle instansene av klassen. Man kan skriva `static` før medlemsvariabelen for å sikre seg at det bare finnes en kopi av variabelen når programmet blir kjørt, istedenfor en kopi per objekt som lages i klassen.

`inline` trengs for å kunne initialisere statiske variabler i klassesdefinisjonen. Å deklare en variabel som `constexpr` gir kompilatoren beskjed om at variabelen skal evalueres ved kompilering. Det gir oss også muligheten til å bruke variabelen i `constexpr`-funksjonar som kan evalueres ved kompilering. En variabel som er `static constexpr` er derfor felles for alle instansene av klassen, og evalueres ved kompilering. Dette betyr også at den kan (og må) initialiseres direkte i `.h`-fila til klassen. På denne måten oppnår man raskere kjøring av koden, fordi den ene kopien av variabelen allerede er evaluert ved kompilering og trengs aldri å evalueres igjen.

Nyttig å vite: Kort om callback

En callback-funksjon er en funksjon blir kalt når du trykker på en knapp i GUI-et. Den skal ikke ta inn noen parametere og returnerer `void`. For å koble en callback-funksjon til en knapp bruker vi `setCallback()`-funksjonen til knappen. Les mer om callback-funksjoner [her](#).

- d) **Deklarer og definer en callback-funksjonen `void cb_quit()`.** Denne callback-funksjonen skal brukes av avslutnings-knappen, så derfor må den kalle på den arvede medlemsfunksjonen `close()` som lukker vinduet.

- e) **Legg inn et `TDT4102::Button`-objekt, `quitBtn`, som privat medlemsvariabel.** `quitBtn` må konstrueres i initialiseringslista til `MeetingWindow`. `quitBtn` legges til vinduet med `add(quitBtn)` i funksjonskroppen til `MeetingWindow`-konstruktøren. `quitBtn` skal bruke `cb_quit` som callback-funksjon, og det gjøres slik `setCallback(std::bind(&MeetingWindow::cb_quit, this))` på `quitBtn`. Prøv å kjør programmet og se om det fungerer som forventet.

5 Person-funksjonalitet (20%)

I denne oppgaven skal vi legge til et par widgets til vinduet. Her er det viktig å passe på at ingen av widgets-ene ligger oppå hverandre. Da vil de ikke oppføre seg som forventet.

- a) **Legg til to `TDT4102::TextInput`: `personName` og `personEmail`.**

Disse er to innskrivingsfelt for parameterene til en ny `Person`. Disse må også legges til vinduet på samme måte som `quitBtn`.

- b) Legg til en ny `TDT4102::TextInput`, `personSeats`. Denne skal du bruke for å gi personene en bil i `newPerson`. Dersom `personSeats` er et tall som er større enn null lager du et `Car`-objekt og gir pekeren til denne til `Person`- konstruktøren. Sjøføren må først ha plass, så du må også "reservere" et sete i `Car`-objektet! *Tips: Du kan konvertere fra `string` som du får fra `getText()` på `personSeats`, til `int` ved hjelp av funksjonen `stoi("en streng")`. Den konverterer "en streng" til heltall. Dersom strengen ikke er et tall vil funksjonen gi en feilmelding. Det kan derfor være lurt å legge inn unntakshåndtering for tilfellene der brukeren ikke skriver inn et tall.*

- c) Legg inn `vector<shared_ptr<Person>>` `people` som en medlemsvariabel, og så definer og implementer en ny funksjon, `void newPerson()`.

Denne vektoren skal inneholde pekere til alle personer som blir lagt til gjennom tekstboksene.

`newPerson` skal lese det som er skrevet inn i tekstboksene og legge til en ny person i vektoren med disse argumentene. Dette skal være et anonymt/navnløst objekt, så her må du bruke `new`:

```
people.emplace_back(new Person{/*Dine argument*/});
```

For å hente innholdet i tekstboksene, må du kalle medlemsfunksjonen `getText()` på innskrivings-feltene. Sjekk også om en av parameterene mangler, slik at det ikke legges til ufullstendige personer. Husk også å tømme tekstboksene hver gang du legger til en person, ved å kalle funksjonen `setText()`.

- d) Legg til en ny `Button`, `personNewBtn` med en tilhørende `callback`-funksjon, `cb_new_person()`.

`Callback`-funksjonen skal kalle `newPerson()`.

- e) Test om programmet fungerer som forventet.

En enkel måte å gjøre dette på er å lage en `public` funksjon som printer alle personene i `people`-vektoren. Denne funksjonen kan du kalle i `main()`.

6 Utvidelse av GUI (Frivillig)

I denne oppgaven kan du fullføre GUI-et for samkjøringen. Det skal nå være to sider: en for `Person` og en for `Meeting`. Man skal kunne se alle møter/personer som er lagt inn og kunne legge inn nye. Vinduet skal ha en knapp for å avslutte, to knapper som respektivt bytter til `Meeting`- og `Person`-siden, et tekstfelt for informasjon, et felt for hver parameter å legge inn, og to knapper som respektivt legger til en ny `Person` eller `Meeting`. I tillegg skal det være to felt der du kan velge en person å legge til et spesifikt møte, og en knapp som utfører dette.

- a) **Legg til en ny `TDT4102::TextInput`, `personData`, som privat medlem.**

Dette skal være et større tekstfelt der det er plass til flere linjer. Det skal fungere som display i vårt GUI, og skal vise personene som har blitt lagt inn.

- b) **Lag to nye funksjoner, `void showPersonPage()` og `void showMeetingPage()`.**

Disse skal vi bruke til å bytte mellom `Person` og `Meeting` sidene. For å gjøre dette, må `showPersonPage()` kalle medlemsfunksjonen `setVisible(true)` på alle elementer som er knyttet til den siden, mens `showMeetingPage()` må kalle `setVisible(false)`. Det omvendte gjelder for alle kommende elementer som er knyttet til `Meeting`-siden.

- c) **Legg til to `TDT4102::Buttons`, `createPersonButton`, og `createMeetingButton`, med tilhørende callback-funksjoner, `cb_persons()` og `cb_meetings()`.**

Callback-funksjonene skal respektivt kalle `showPersonPage()` og `showMeetingPage()`.

- d) **Legg inn fire nye `TDT4102::TextInput` til `Meeting`-sida: `meetingSubject`, `meetingDay`, `meetingStart` og `meetingEnd`.**

Disse skal vi senere bruke for å legge inn `Meeting`-objekt. Nå er det lurt å kalle `showPersonPage()` på slutten av `MeetingWindow`-konstruktøren.

- e) **Legg inn to `TDT4102::DropDownList` til `Meeting`-sida: `meetingLocation` og `meetingLeader`.**

`TDT4102::DropDownList` lager en rullegardinliste. For å legge til et valg må du kalle funksjonen `add` med en strengparameter. Legg inn de tre relevante stedene til `location` rullegardinlista `meetingLocation` vha. `MeetingWindow`-konstruktøren. Utvid `newPerson` til at navnet til den nye personen legges til som et valg i `meetingLeader`. Dette kan gjøres ved å bruke funksjonen til `TDT4102::DropDownList` som heter `setOption()`. Denne tar inn en parameter av typen `vector<string>` med alle valg som skal vises i rullegardinlista. Denne funksjonen vil oppdatere rullegardinlista til å vise valgene som står i `vector`-en.

- f) **Legg inn en ny funksjon, `void newMeeting()`, en callback som kaller denne, `cb_new_meeting()`, og en knapp med denne callbacken, `meetingNewBtn`.**

`newMeeting` skal lese parametere fra feltene og konstruere et `Meeting` i en ny `vector<unique_ptr<Meeting>>`, `meetings`. `TDT4102::DropDownList` har medlemsfunksjonen `getValue()` som returnerer posisjonen til valget i lista, så bruk dette for å finne rett sted eller rett møteleder i vektoren over personer. Husk å sjekke om parameterene er gyldige.

- g) **Lag et display. `meetingData`** Dette displayet skal vise alle møtene som er lagt til i GUI.