



Norges teknisk–naturvitenskapelige
universitet
Institutt for datateknologi og
informatikk

TDT4102 Prosedyre-
og objektorientert
programmering
Vår 2022

Øving 9

Frist: 2022-03-25

Mål for denne øvinga:

- Lære om operatorar og interaksjon mellom klasser av ulike typar.
- Lære å implementere og å bruke klasser.
- Lære å bruke enkle klasser for eit enkelt grafisk brukargrensesnitt (GUI).
- Lære å bruke peikarar
- Lære å bruke `unique pointer` og `shared pointer`

Generelle krav:

- Bruk dei eksakte namn og spesifikasjonar gjeve i oppgåva.
- Teorioppgåver svarar du på med kommentarar i kildekoden slik at læringsassistenten enkelt finn svaret ved godkjenning.
- 70% av øvinga må godkjennast for at den skal vurderast som bestått.
- Øvinga skal godkjennast av stud.ass. på sal.
- Det anbefalast å nytte ein programmeringsomgjevnad(IDE) slik som Visual Studio Code.

Tilrådd lesestoff:

- Kapittel 17, 19.5.4 og 19.5.5 i PPP

DEL 1: NTNU-samkøyring (60%)

Du har fått sommarjobb i det nye studentføretaket «EcoTrans» som er starta av nokre miljømedvitne NTNU-studentar. Omorganiseringa til nye NTNU har skapt eit behov for koordinert transport mellom byane Trondheim, Ålesund og Gjøvik. Kvar veke reiser mange tilsette og studentar mellom desse byane, ofte i privatbilar med ledige sete. EcoTrans vil lage eit enkelt datasystem for å stimulere til miljøvenleg samkøyring, og i denne oppgåva skal du skrive nokre kodebitar for eit slikt system.

1 Car-klassa (10%)

a) Deklarer ei klasse Car.

Car skal ha eit heiltal `freeSeats` som privat medlemsvariabel som indikerer kor mange ledige sete det er i bilen. Car skal også ha to `public` medlemsfunksjonar `hasFreeSeats` og `reserveFreeSeat`. `hasFreeSeats` returnerer `true` om bilen har ledige seter, og `false` elles. `reserveFreeSeat` «reserverer» eit ledig sete ved å dekrementere `freeSeats`-variabelen (du kan gå ut frå at funksjonen berre verte kalla på om det er ledige sete).

Deklarasjonar for medlemsfunksjonane:

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

Nyttig å vite: Const correctness

Det er god praksis å markere medlemsfunksjonar som ikkje endrar objektet med `const`. Dette gjer det enklare å finne feil i koden, og let oss bruke medlemsfunksjonane sjølv om objektet er konstant.

<pre>class NumberClass { int number; public: NumberClass(int number) : number{number} {} // markert const int getNumber() const { return number; } // ikkje 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 // IKKJE OK! // Kompileringsfeil: // kan ikkje kalle ein funksjon // som ikkje er markert const, // på eit const objekt y.setNumber(j+1); }</pre>
---	---

b) Deklarer og implementer ein konstruktør som tek inn kor mange ledige sete bilen har.

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

Hugs at deklarasjonen skal vere i `Car.h`, og definisjonen (implementasjonen) skal vere i `Car.cpp`.

2 Person-klassa (20%)

a) Deklarer ei klasse `Person`.

Denne skal ha dei private medlemsvariablane `name` og `email`, begge av typen `string`. I tillegg skal klassa ha ein `unique_ptr` `car`, som er ein peikar til ein `Car`. Legg merke til at vi ønskjer å bruke ein peikar og ikkje referanse til `Car`-objektet.

Grunnen til at vi ønskjer å bruke ein peikar er fordi ein peikar kan ha verdien `nullptr` og det passar fint for å representere at ein person *ikkje* har bil. Om vi nytta referanse i staden for peikar ville det vorte vanskelegare å representere dette. Dette er godt forklart i læreboka §17.9.1, og mot slutten av det avsnittet er det oppsummert når ein anbefaler pass-by-value, peikar, eller referanse-parameter.

Klassa skal ha ein konstruktør som set `name`, `email` og `car` til verdier gjeve av parameterlista. For `car` nyttar vi `nullptr` som eit såkalla «default argument» (standard-verdi). Det tyder at konstruktøren kan brukast med berre de to første parametrane, og då vil den tredje få denne standard-verdien. Sjå også nyttig-å-vite-boks om dette temaet. Deklarer ein `get`-funksjon både for `name` og `email`. Deklarer også ein `set`-funksjon for `email`.

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

Kva er ein `std::unique_ptr`? `std::unique_ptr` er ein smartpeikar som *eig* og handterer eit anna objekt gjennom ein peikar, og slettar automatisk minnet som tilhøyrrer objektet når `std::unique_ptr`-instansen blir destruert. Når ein `unique_ptr` blir oppretta kallar han automatisk `new`, og når han går ut av skop kaller han sjølv `delete`. Dette vil seie at vi slepp å bruke `new` og `delete` når vi skal bruke dynamisk allokerde objekt. Merk at dette faget berre er ein liten intro til det man kan gjere med `std::unique_ptr`.

Korleis opprette ein `std::unique_ptr`? C++14-funksjonen `std::make_unique` brukes for å lage eit nytt `std::unique_ptr`-objekt. Han allokterer òg nødvendig minne til objektet `std::unique_ptr` handterer.

```
/* Her opprettast ein unique_ptr som handterer eit Student-objekt.
   Argumenta til funksjonen blir gitt direkte til Student-konstruktøren. */
unique_ptr<Student> s1 = make_unique<Student>("daso", "daso@stud.ntnu.no");
/* Vi kan òg bruke auto når vi opprettar ein unique_ptr. */
auto s2 = make_unique<Student>("lana", "lana@stud.ntnu.no");
```

Korleis bruke ein `std::unique_ptr`? Derefereringsoperatoren (*) og piloperatoren (->) blir brukt som om det er ein vanleg peikar.

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

Ein `unique_ptr` *eig* objektet det peikar til (berre ein `unique_ptr` kan peike på eit objekt om gangen) — den tillèt ikkje peikaren å bli kopiert. (Viss vi hadde kunnet kopiere ein `unique_ptr` og ein av peikarane hadde gått ut av skop, ville objektet vorte sletta, og dei gjenverande `unique_ptr` ville peika til minne vi ikkje veit kva inneheld.) Derimot kan eigarskapet overførast vha. `std::move`. Vi seier at `std::move` *overfører eigarskapet* av objektet.

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

Etter denne operasjonen er `s3` ein `unique_ptr` som eig det `Student`-objektet som `s1` tidlegare eigde. `s1` er derimot i ein «gyldig men uspesifisert» tilstand.

Nokre gangar ønskjer vi å la andre bruke objekta som blir peika til av `unique_ptr`-instansen utan å overføre eigarskapet. Vi kan da få tak i den underliggande peikaren, vha. medlemsfunksjonen `get`.

```
void printStudent(Student* sPtr);
printStudent(s2.get()); /* s2.get() returnerer den underliggande
    peikaren.*/
```

Medlemsfunksjonen `get` kan òg bli brukt til å sjekke om ein `unique_ptr` har eit tilknytta objekt, ved å samanlikne med `nullptr`.

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

`unique_ptr` kan verke vanskeleg, men ikkje overkompliser det! Tenk på det som ein vanleg peikar, bare at han eig objektet han peikar på og dermed har ansvar for å fiks minnet til objektet sjølv. Og siden det blir krøll viss fleire skal eige det same objektet, kan ein ikkje ha meir enn éin `unique_ptr` til eit objekt, så dersom ein anna `unique_ptr` skal peike dit må ein «move» eigarskapet.

`std::shared_ptr` fungerer på mange måtar likt, men objektet han peikar på kan nå bli eigd av fleire `std::shared_ptr`s. For kvar peikar du da har til objektet vil ein teljar legge til éin. Viss peikarane går ut av skop, eller blir fjerna, vil teljaren reduserast med éin, og først når den siste `std::shared_ptr`-en går ut av skop, blir objektet sletta.

b) Implementer konstruktøren og get-/set-funksjonane frå førre deloppgåve.

Bruk initialiseringsliste i konstruktøren.

Nyttig å vite: default arguments

For å unngå at ein skal definere fleire ulike funksjonar som gjer det same, men har ulik tal på parametrar i parameterlista, så finnst det *default arguments*.

Til dømes kan ein funksjon som alltid skal leggje saman to tal vere standardisert til å inkrementere det første argumentet med ein, dersom det andre argumentet ikkje er oppgjeve. I staden for å lage to funksjoner som gjer same arbeid eller kallar på ein annan, er det formålstenleg å samle dei. Dette kan ein også bruke med medlemsfunksjonar i klassar.

```
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 skrivast i deklarasjonen, men ikkje i definisjonen. Argumenta som har default-verdi må også skrivast sist i parameterlista.

c) Lag medlemsfunksjonen `hasAvailableSeats()`.

Funksjonen returnerer `true` om personen eig ein bil og bilen har ledige sete.

d) Overlast operator<<, som skal skrive ut innhaldet i `Person` til ein ostream.

Drøft:

- Kvifor bør denne operatoren deklarerast med `const`-parameter? (t.d. `const Person& p`)
- Når bør vi, og når bør vi ikkje (ev. kan ikkje) nytte `const`-parameter?

e) Skriv testar for `Person` i `main()`.

Opprett fleire personar, og prøv å teste ulike tilfelle (t.d. har personen bil? Kva med når personen ikkje har bil?).

Tips: Viss vi opprettar ein `shared_ptr` til ein `Person`, må vi lage eit `Person`-objekt ved å bruke `new` og ikkje `make_shared`. `make_shared` kan berre brukast når objektet har ein kopikonstruktør. `unique_ptr` har ikkje ein kopikonstruktør, og ein `unique_ptr` som medlemsvariabel gjer at default kopikonstruktøren til klassen slettast.

3 Meeting-klassa (30%)

a) Deklarer ein `scoped enum`, med namn `Campus`, som inneheld verdiar for dei ulike byane (Trondheim, Ålesund og Gjøvik). Overlast operator<< for `Campus`, som skal skrive campusnamnet til ein ostream.

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

b) Definer klassa `Meeting`.

Klassa skal ha følgjande private medlemsvariablar:

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

Implementer `get`-funksjonar for `day`, `startTime`, `endTime`, `location`, `subject`, og `leader` som ein del av klassesdefinisjonen.

c) Lag medlemsfunksjonen `addParticipant`.

Den skal ta inn ein `std::shared_ptr` til eit `Person` objekt og leggje den inn i `participants`.

d) Lag ein konstruktør for `Meeting`-klassa som tek inn `day`, `startTime`, `endTime`, `location`, `subject`, og `leader`.

Hugs at møteleiaren også er ein deltakar.

e) **Teori:** Når vi opprettar eit `Meeting`-objekt, korleis blir det allokerde minnet rydda opp når vi slettar objektet eller stoppar programmet?

f) Lag funksjonen `getParticipantList()`.

Dette skal vere ein medlemsfunksjon i `Meeting`. Funksjonen har ingen parametrar og returnerer ein `vector<string>` med namn på deltakarane.

g) Overlast operator<< for Meeting.

Denne operatoren skal IKKJE vere ein `friend` av `Meeting`. Du står fritt til å velje format sjølv, men du skal skrive ut `subject`, `location`, `startTime`, `endTime`, og namnet på møteleiaren. I tillegg skal han skrive ut ei liste med namna på alle deltakarane.

Test funksjonen din frå `main()`.

h) Skriv funksjonen `findPotentialCoDriving`.

Dette skal vere ein medlemsfunksjon i `Meeting`. Funksjonen skal ta inn eit anna møte, og skal returnere ein vektor med `Person`-peikarar. Vektoren skal bestå av alle personar som:

- har ledige plassar i bilen sin og
- skal til eit anna eller same møte, som
 - er på same stad som `this`-møtet,
 - er på same dag som `this`-møtet,
 - har start-tid som er mindre enn ei time forskjellig frå start-tida til `this`-møtet, og
 - har slutt-tid som er mindre enn ei time forskjellig frå slutt-tida til `this`-møtet.

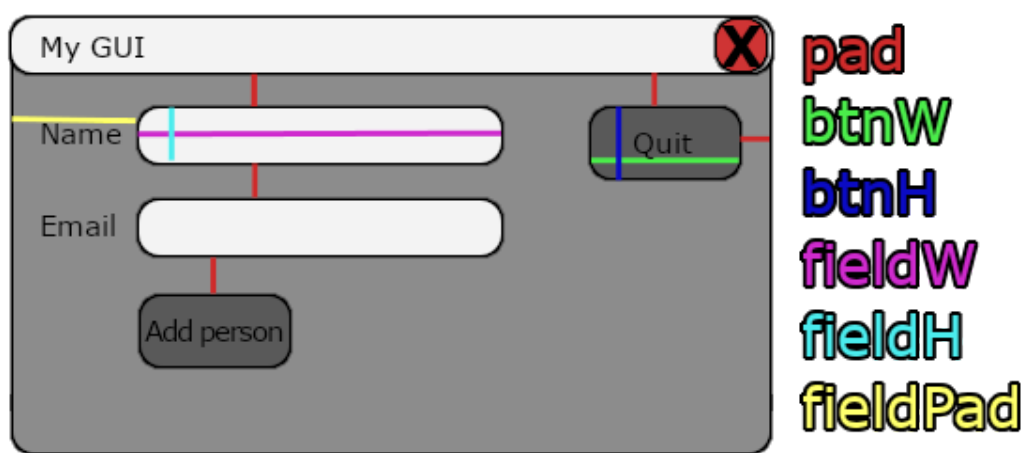
Hint: Funksjonen vil ha følgjande returtype: `vector<shared_ptr<Person>`.

Hugs `const correctness`.

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

For å gjere programmet meir brukarvenleg vil dykk lage eit grafisk brukargrensesnitt (GUI) der passasjerar kan melde seg inn. Den skal ha to tekstfelt, eit for å skrive inn passasjerens namn og eit for e-post. Det skal også vere to knappar: ein for å leggje til personen og den andre for å avslutte programmet.

Før ein går laus på kodinga til eit GUI, er det veldig lurt å danne seg ei skisse for korleis vindauget skal sjå ut. Det er også lurt å namngje alle ulike avstandar i vindauget. Dette er delvis slik ein ikkje treng å sjonglere alle dei ulike verdiane i hovudet, men mest fordi ein då kan endre på alle felles avstandar på ein stad. Under kjem ei mogeleg skisse av GUI-et der kvar farge er kopla til ein variabel. Det er ikkje eit krav at du følgjer denne skissa, men alle elementa skal vere med til slutt.



Alle variablar og funksjonar i resten av oppgåvene skal vere medlem av vindaug-klassa vi lagar.

4 Oppretting av GUI (20%)

Til denne oppgåva skal vi bruke `AnimationWindow` slik vi har gjort før, noko som gjer at vi overlet programstyringa til vindauget sjølv. I praksis tyder det at vi må lage vår eiga vindaug-klasse som arvar frå `AnimationWindow`, og at klassa skal styre programmet.

Vidare skal vi nytte FLTK sine widgets `Fl_Input` og `Fl_Button`.

- Lag ei ny klasse `MeetingWindow`, som arvar public frå `AnimationWindow`, og konstruktøren `MeetingWindow(int x, int y, int w, int h, const string& title)`.**
Plasser klassesdeklarasjonen i `MeetingWindow.h`. Sidan `AnimationWindow` ikkje har ein default-konstruktør, så må du kalle `AnimationWindow`-konstruktøren i initialiseringslista, der argumenta skal vere dei du tok inn i `MeetingWindow`-konstruktøren. La konstruktør-kroppen stå tom inntil vidare.
- Lag eit `MeetingWindow`-objekt i `main()` og kall `gui_main()`.**
Lag eit objekt av typen `MeetingWindow` og legg deretter inn eit kall på `gui_main()` i `main()`. `gui_main()` overlet programstyringa til vindauget du har laga. Prøvekøyr koden, du skal få opp eit blankt vindaug.
- Før du byrjar på å leggje inn element, er det lurt setje inn nokre heiltal i klassa som definerer oppsettet i vindauget.** Sjå skissa over for eit forslag. Sidan desse ikkje skal endrast og er definerte før kompileringa, er det lurt å deklarere dei `static constexpr int` og gi dei verdiar direkte i `.h` fila.

Nyttig å vite: static og static constexpr

Ein statisk medlemsvariabel er ein variabel som er felles for alle instansane av klassa. Ein kan skriva `static` før medlemsvariabelen for å sikre seg at det berre finnast ein kopi av variabelen når programmet blir køyrt, i staden for ein kopi per objekt som lagast i klassa.

`inline` trengst for å kunna initialisera statiske variablar i klassesdefinisjonen.

Å deklarera ein variabel som `constexpr` gir kompilatoren beskjed om at variabelen skal evaluerast ved kompilering. Det gir oss også moglegheita til å bruke variabelen i `constexpr` funksjonar som kan evaluerast ved kompilering. Ein variabel som er `static constexpr` er difor felles for alle instansane av klassa, og evaluerast ved kompilering. Dette betyr også at den kan (og må) initialiserast direkte i `.h`-fila til klassa. På denne måten oppnår ein raskere køyring av koden, fordi den eine kopien av variabelen allereie er evaluert ved kompileringa og treng aldri å evaluerast igjen.

Nyttig å vite: Kort om callback

Ein callback-funksjon vert kalla av ein knapp i GUI-et når du trykkjer på den og har signaturen `void cb_my_callback(Fl_Widget*, void* pw);`. Den første parameteren bryr vi oss ikkje om, men den andre er adressa til GUI-vindauget. For at du skal kunne bruke vindauget, må du først fortelje funksjonen at den må tolke `pw` som ein `MeetingWindow`-referanse. Dette gjer du med å kalle `static_cast<MeetingWindow>(pw)`, og deretter kan du kalle medlemsfunksjonar med vanleg dot-notasjon, slik som vist under.

```
void cb_my_callback(Fl_Widget*, void* pw){
    static_cast<MeetingWindow*>(pw)->member_func();
}
```

d) Deklarer callback-funksjonen `void cb_quit(Fl_Widget*, void* pw)`.

Ein callback-funksjon må deklarerast som ein `static` medlemsfunksjon til `MeetingWindow`, dette gir GUI-et moglegheita til å kalle funksjonen uavhengig av eit objekt.

e) Implementer `cb_quit()` frå forrige deloppgåve.

Denne callback-funksjonen skal nyttast av avslutnings-knappen, så derfor må den kalle den arva medlemsfunksjonen `hide` som avsluttar vindauget.

f) Legg inn eit `Fl_Button`-objekt, `quitBtn`, som privat medlemsvariabel.

`quitBtn` må konstruerast i initialiseringslista til `MeetingWindow`. `quitBtn` legges til vindauget ved å bruke `add(quitBtn)` i funksjonskroppen til `MeetingWindow`-konstruktøren. `quitBtn` skal bruke `cb_quit` som callback-funksjon, og det gjerast ved å kalle `callback(cb_quit, this)` på `quitBtn`. Prøv å køyre programmet her og sjå om det funkar som forventa.

5 Person-funksjonalitet (20%)**a) Legg til to `Fl_Input`: `personName` og `personEmail`.**

Desse er to innskrivingsfelt for parametrane til ein ny `Person`. Desse må også leggest til vindauget på same måte som `quitBtn`.

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

Denne vektoren skal innehalde peikarar til alle personar som vert lagt til gjennom tekstboksane.

`newPerson` skal lese det som er skrive inn i tekstboksane og leggje til ein ny person i vektoren med desse argumenta. Dette skal vere eit anonymt/namnlaust objekt, så her må du bruka `new`:


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

For å hente innhaldet i tekstboksane, må du kalle funksjonen `value()` på innskrivingsfeltet. Sjekk også om ein av parametranne manglar, slik at det ikkje leggjast til ufullstendige personar. Hugs også å tømme tekstboksane kvar gang du leggjer til ein person, ved at `value()` også kan brukast til å setje tekstboksane til ein verdi.

- c) Legg til ein ny `F1_Button`, `personNewBtn` med ein tilhøyrande callback-funksjon, `cb_new_person()`.

Callback-funksjonen skal kalle `newPerson`.

- d) Test om programmet fungerer som venta.

Ein enkel måte å gjere dette på er å lage ein `public` funksjon som printar alle personane i `people`-vektoren. Denne funksjonen kan du kalle i `main()` etter `gui_main()`.

6 Utviding av GUI (Frivillig)

I denne oppgåva kan du fullføre GUI-et for samkøyringa. Det skal no verte to sider: ei for **Person** og ei for **Meeting**. Ein skal kunne sjå alle møte/personar som er innførte og kunne leggje inn nye. Vindauget skal ha ein knapp for å avslutte, to knappar som respektivt byttar til **Meeting**- og **Person**-sida, eit tekstfelt for informasjon, eit felt for kvar parameter å leggje inn, og to knappar som respektivt legg til ein ny **Person** eller **Meeting**. I tillegg skal det vere to felt der du kan velje ein person å leggje til eit spesifikt møte, og ein knapp som utfører dette.

a) Legg til ein ny `Fl_Input`, `personSeats`.

Denne skal du bruke for å gi personane ein bil i `newPerson`. Dersom `personSeats` har eit tal som er større enn null lagar du eit `Car`-objekt og gir peikaren til denne til `Person`-konstruktøren. Sjøføren må først ha plass, så du må også "reservere" eit sete i `Car`-objektet! *Tips: Du kan gjere om frå `string` som du får frå å kalle `value()` på `personSeats`, til `int` ved hjelp av funksjonen `stoi("ein streng")`. Den konverterer "ein streng" til heiltal.*

b) Legg til ein `Fl_Multiline_Output`, `data`, som privat medlem.

Dette er ein `Fl_Output` som kan vise fleire linjer. Denne skal fungere som display i vårt GUI.

c) Lag to nye funksjonar, `void showPersonPage()` og `void showMeetingPage()`.

Desse skal vi bruke til å bytte mellom `Person` og `Meeting` sidene. For å gjere dette, må `showPersonPage()` kalle medlemsfunksjonen `show()` på alle element som er knytta til den sida, medan `showMeetingPage()` må kalle `hide()`. Det omvendte gjeld for alle komande element som vert knytta til `Meeting`-sida. `data` kan vere felles, men då må du kalle `displayPeople()` i `showPersonPage()`.

d) Legg til to `Fl_Buttons`, `createPersonButton` og `createMeetingButton`, med to tilhøyrande callback-funksjonar, `cb_persons()` og `cb_meetings()`.

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

e) Legg inn fire nye `Fl_Input` til `Meeting`-sida: `meetingSubject`, `meetingDay`, `meetingStart`, og `meetingEnd`.

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

f) Legg inn to `Fl_Choice` til `Meeting`-sida: `meetingLocation` og `meetingLeader`.

`Fl_Choice` lagar ei rullegardinliste. For å leggje til eit val må du kalle funksjonen `add` med ein strengparameter. Legg inn dei tre relevante stadane til `location` i `MeetingWindow`-konstruktøren. Utvid `newPerson` til at namnet til den nye personen leggst til som eit val i `meetingLeader`.

g) Legg inn ein ny funksjon, `void newMeeting()`, ein callback som kallar denne, `cb_new_meeting()`, og ein knapp med denne callbacken, `meetingNewBtn`.

`newMeeting` skal lese parametrar frå felte og konstruere eit `Meeting` i ein ny `vector<unique_ptr<Meeting>, meetings`. `Fl_Choice` har medlemsfunksjonen `value()` som returnerer posisjonen til valet i lista, så bruk dette for å finne rett stad eller rett møteleiar i vektoren over personar. Hugs å sjekke om parametraner er gyldige.

h) Lag ein funksjon som legg inn alle møta til displayet `data`.

Kall på denne i `newMeeting()` og `showMeetingPage()`.