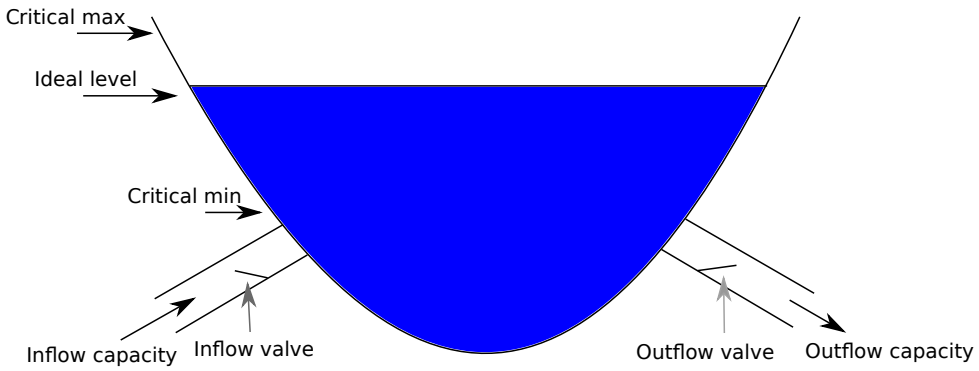
Source of Life

I denne oppgaven ser vi på en simuleringsmodell av et system for styring og overvåking av demninger som gir vann til det kommunale vannanlegget.



Figur 1: Schematic of a dam with the model parameters included in our simulation model

Hensikten med systemet er å overvåke vannstanden til demninger og justere inn- og utstrømningsventiler etter behov for å opprettholde vann på et ideelt nivå. Som en forenkling, anta at inn- og utstrømningsrørene aldri går helt tom og derfor kan levere eller forbruke så mye vann det er behov for. De har derimot en begrenset flytkapasitet. Dette betyr at bare en begrenset mengde vann kan strømme inn eller ut fra demningen på et gitt tidspunkt, og det tar derfor tid å gjenopprette vannstanden til normalen etter deteksjon av et overskudd eller mangel på vann. Figur 1 viser skjematisk en dam med nevnte modellparametere.

I stedet for målinger fra faktiske sensorer, er modellen her et simuleringsscenario bestående av en serie av diskrete tidspunkter. For hvert tidspunkt kan simuleringen motta sensoravlesninger som indikerer at vannstanden til en demning har endret seg, og den kan da utføre ventiljusteringer for å bringe vannnivået i systemet nærmere det ideelle nivået igjen. [1]

Model parameters

Dette avsnittet beskriver i mer detalj parametrene over, og presist hvilken kontekst de brukes i (jf. Figur 1).

Inflow capacity How much water that can flow into a dam during a timestep

Ideal level The ideal level of the dam. When the simulation starts, the dams are at this level.

Critical high The critical high level of the dam. An alarm is raised if the dam fills above this level.

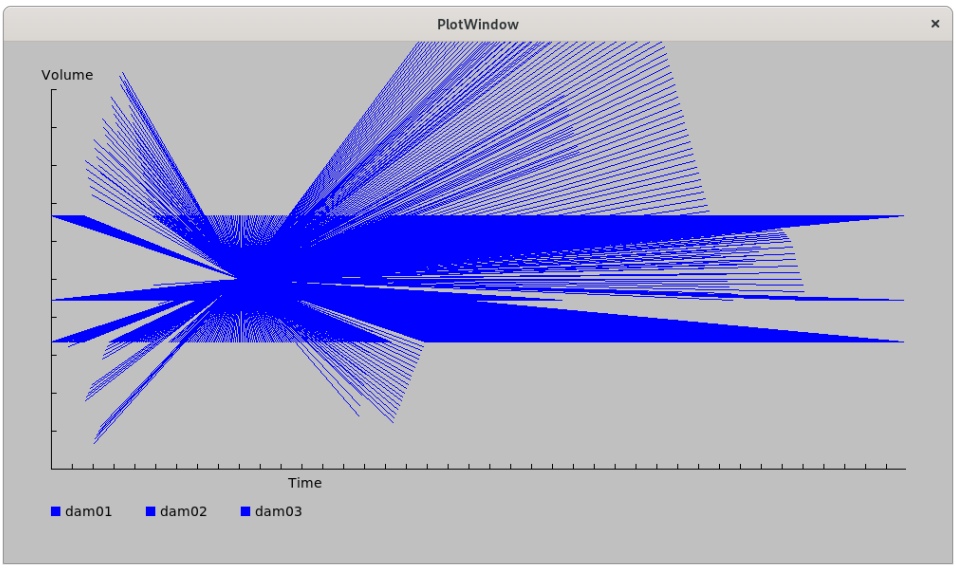
Critical low The critical low level of the dam. An alarm is raised if the dam empties below this level.

The three different dams to simulate in this exam have parameters as shown in Tabell 1.

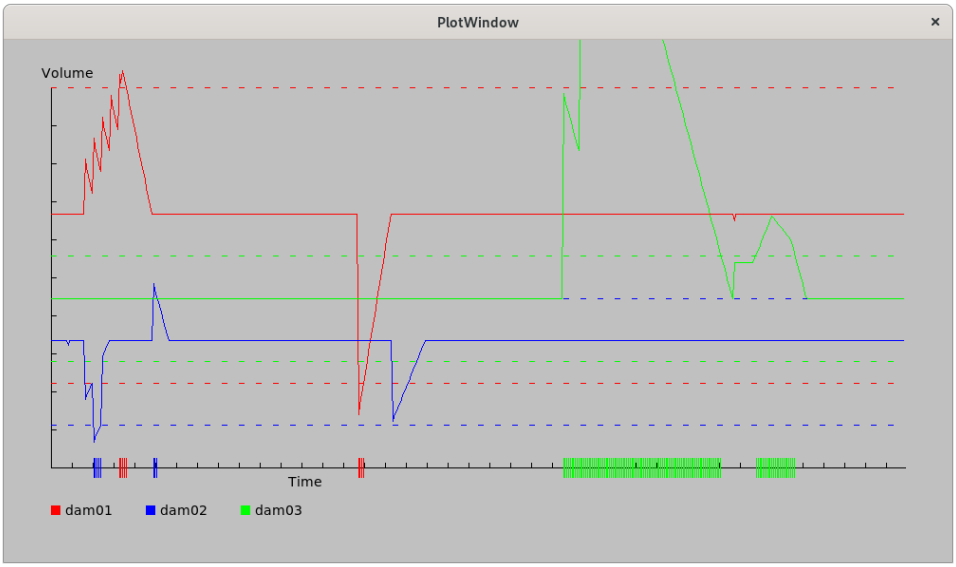
Name	Ideal level	Critical high	Critical low	Outflow capacity	Inflow capacity
dam01	1200	1800	400	40	50
dam02	600	800	200	30	20
dam03	800	1000	500	30	40

Tabell 1: Parameters for the simulated dams.

[1] Dette ligner et virkelig distribusjonsystem hvor det kommer ekte sensoravlesninger med en bestemt frekvens.



Figur 2: The initial plot produced by the handed-out code. This is not very useful.



Figur 3: An example of a plot produced by the code after the plotting (P) functions have been implemented. The colored lines intersecting the x-axis indicates alarms for levels exceeding the critical maximums and minimum limits. The dashed horizontal lines indicate the limits.

Implementation overview

Implementasjonen av denne oppgaven er delt inn i flere forskjellige filer, men du trenger bare å endre noen av dem for å løse oppgavene. En betydelig del av koden er bare et stillas som kreves for å få programmet til å kjøre. Husk dette, da det ikke er lurt å bruke tid på å forstå hele kodebasen, siden det ikke er nødvendig for å kunne løse oppgaven. For å gjøre det lettere å navigere i koden vil hver enkelt deloppgave fortelle deg hvilke filer du trenger å endre og hvilke deklarasjoner du trenger å forstå for å løse den deloppgaven.

Første gang du kompilerer og kjører programmet skal du se vinduet vist i Figur 2. Det er klart at plottet ikke er veldig nyttig, men ved å løse oppgavene, vil du gradvis implementere de manglende delene av programmet og jobbe mot et ønsket resultat som vist i Figur 3. Det er ikke et krav for å bestå eksamen at all innlevert kode faktisk kjører. Vi anbefaler likevel at du tester svarene dine for å bekrefte at de er korrekte.

De individuelle deloppgavene som utgjør hele denne tredje oppgaven sier alle hvilke kildefiler som må

endres for å løse den deloppgaven. I kildefilene vil du finne deloppgavene merket med en kommentar:

`// Assignment XX`

etterfulgt av beskrivelsen av oppgaven (på engelsk). Disse beskrivelsene er identiske med tilsvarende oversatte beskrivelser du vil finne i dette dokumentet. Du vil også se kommentarer som sier

`// Used for assignment XX.`

Disse kommentarene brukes til å trekke din oppmerksomhet mot implementasjonsdetaljer, f.eks. deklarasjoner, som er nødvendige for å løse en bestemt oppgave. Til slutt, i hver oppgave er det et matchende par kommentarer

`// BEGIN: XX` og `//END: XX`.

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 P1 ser du følgende kode i utdelte `plotter.cpp`

```
Color::Color_type GraphColor::next_color()
{
  // BEGIN: P1
  (void)cur_color;
  return Color::black;
  // END: P1
}
```

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

```
Color::Color_type GraphColor::next_color()
{
  // BEGIN: P1

  // Your code here

  // END: P1
}
```

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

Theme 1: Plotting

Alle oppgavene i denne Plotting-delen gjelder plottfunksjonaliteten implementert i filene `plotter.h` og `plotter.cpp`

3.1 P1: next_color

Implementer funksjonen `next_color` slik at hvert kall til funksjonen returnerer sekvensielt neste farge fra vektoren `colors` definert i klassen `Color` (`plotter.h`). Når det ikke er flere farger i listen, skal funksjonen kaste et unntak med en beskrivende melding. Bruk den private klassemedlemsvariabelen `cur_color` for å holde oversikt over gjeldende posisjon i `colors` vektoren.

Eksempel: Det første kallet til `next_color` returnerer `Color::red` som er den første i vektoren `colors`. Det andre kallet returnerer `Color::blue` som er nummer to i listen, og så videre.

3.2 P2: Random color

Klassen `GraphColorRandom`, definert i `plotter.h`, arver fra klassen `GraphColor` og endrer funksjonaliteten slik at dens `next_color`-medlemsfunksjon nå returnerer fargene i en tilfeldig rekkefølge. Merk at hver farge fortsatt bare skal returneres en gang. Implementer konstruktøren til `GraphColorRandom`-klassen slik at den tilfeldig endrer rekkefølgen i `colors`-vektoren når klassen blir instansiert.

For å teste implementasjonen din, endre instansieringen av `GraphColor colors` i klassedefinisjonen av `Plotter` i `plotter.h` som angitt i den filen.

Implementer `GraphColorRandom` klassen.

Hint: En vanlig algoritme for å stokke en liste tilfeldig er Fisher-Yates[2] gjengitt nedenfor:

```
for i from 0 to n-2 do
    j ← random integer such that i <= j < n
    exchange a[i] and a[j]
```

3.3 P3: get_color

Implementer funksjonen `get_color` som tar navnet til en dam som parameter og returnerer plottfargen som er tilordnet den dammen. Funksjonen skal slå opp fargen i klassemedlems-map `dam_colors` definert i `plotter.h`. Hvis en demning ikke finnes i `dam_colors`, skal funksjonen opprette en plass for demningen og tilordne en farge ved å bruke `colors.next_color()`-funksjonen. `colors` er `GraphColor`-instansen som er definert i klassen `Plotter` i `plotter.h`

3.4 P4: Track last point

Plottene av dammenes vannstand er sammensatt av en serie med linjesegmenter. For hver tidssteg får vi bare den nåværende verdien av en gitt dam. Derfor må vi følge med på hvor linjesegmentet for forrige tidssteg stoppet for å vite hvor vi skal starte linjesegmentet for gjeldende tidsperiode. Dette er formålet med `get_last_point`-funksjonen.

Implementer funksjonen `get_last_point` som returnerer det punktet der forrige linje for demningen stoppet. Funksjonen tar to parametre; navnet på dammen og det neste punktet som skal plottes. Alle slike "siste-punkter" skal lagres i klassemedlems-map `last_points` definert i `plotter.h`.

`last_points` knytter navnet til en demning til det siste punktet som foreløpig er plottet for den demningen. Hvis det ikke finnes noen oppføring i `last_points` for demningen, skal funksjonen opprette et nytt innslag for demingen og returne punktet `(0,0)`. Hvis demingen finnes i containeren fra før, skal funksjonen heller returnere det forrige punktet og oppdatere `last_points` med det neste punktet som nå tegnes.

[2] `https://en.wikipedia.org/wiki/Fisher%E2%80%93Yates_shuffle`

3.5 P5: Dam alarms

Hvis en demning havner i en kritisk tilstand, dvs. at vannstanden går utover de kritiske maksimums- og minimumsgrensene, ønsker vi å visualisere dette spesielt i grafen.

Implementer funksjonen `plot_alarm` som plotter en vertikal linje som krysser x-aksen til grafen (som vist i Figur 3) for å indikerer at det oppsto en vannstand-alarm. Bruk funksjonen `get_color` fra oppgave P3 for å fargelegge linjene med riktig farge.

Funksjonen `plot_alarm` tar tre parametere: Navnet på dammen alarmen ble utløst for, tidspunktet som alarmen oppstod på, og typen alarm. Verdien `timestep` brukes til å beregne x-koordinaten for plassering av linjen. Legg merke til at funksjonen `calc_x` finner riktig verdi på x-aksen til koordinatsystem gitt et `timestep`. Alarmen oppgis som en parameter av typen `Dam::WaterAlarm`, en `enum` definert i `dam.h`. Den private medlemsvariabelen `y_len` inneholder høyden på y-aksen. Du kan bruke dette til å bestemme høyden på linjen.

Funksjonen `plot_alarm` blir kalt på hvert `timestep`, og verdien av parameteren `alarm` er returverdien for funksjonen `Dam::get_alarm` (fra oppgave D2) i `dam.cpp`. Du skal bare tegne "alarm-linjen" hvis verdien til alarmparameteren indikerer at det oppsto en alarm på oppgitt `timestep`. Merk at vår default implemenasjon av `get_alarm`-funksjonen produserer noen helt tilfeldige alarmer. Derfor kan du bruke `get_alarm` til testing nå, selv om du ikke har løst oppgave D2 ennå.

Hint: For å få en linjeplassering som krysser x-aksen (som vist i Figur 3) kan du tegne linjer med start- og sluttpunkt

`(calc_x(timestep), y_len + 40)` og `(calc_x(timestep), y_len + 60)`

henholdsvis.

Denne oppgaven skal implementeres i filen `plotter.cpp`.

3.6 P6: Dam limits

For å gjøre det enklere å se når vannstanden til en dam overskrider grensene for maksimal eller minimal vannstand, ønsker vi også å visualisere disse grensene i selve grafen.

Implementer funksjonen `plot_limit` som tegner en stiplet horisontal linje på grafen (noen eksempler ble vist i Figur 3). Funksjonen tar inn navnet til demningen og en grense som parametere, og den kalles når dammene blir initialisert. Husk at du kan regne om en verdi til en y-koordinat ved hjelp av `calc_y`-funksjonen, og at `calc_x`-funksjonen kalt med verdier mellom 0 og `timesteps` vil generere alle de mulige x-koordinatene. Tegn linjene i den fargen som returneres av `get_color`-funksjonen. Til slutt, husk å legge til de `Shapes` du tegner til `shapes`-vektoren slik at de blir deallokert når klassen `Plotter` går ut av skopeet.

Theme 2: Dam models

Oppgavene i Dam-klassen dreier seg om dammodellen implementert i filen `dam.cpp`. Se også definisjonen av klassen `Dam` i `dam.h`

3.7 D1: Water level delta

Implementer funksjonen `get_water_level_delta` som returnerer forskjellen mellom nåværende og ideell vannstand. Funksjonen skal returnere den absolutte verdien av delta (avstanden), så du kan bruke funksjon `abs` til det. Husk at dagens vannstand og den ideelle vannstanden lagres henholdsvis i medlemsvariablene `water_level` og `ideal_level` i klassen `Dam`. Klassen `Dam` er definert i `dam.h`.

3.8 D2: Water level alarms

Implementer funksjonen `get_alarm` som indikerer til kalleren om vannstanden til demningen er over det kritisk høye nivået eller under det kritisk lave nivået. Funksjonen skal returnere en av verdiene som er definert i `WaterAlarm enum` i `dam.h`. Hvis vannstanden er i mellom de kritiske grensene, skal funksjonen også da returner en passende verdi fra samme `enum`.

Husk at reell vannstand, kritisk høy-, og kritisk lav-nivåene er lagret i de private medlemsvariablene `water_level`, `water_level_max` og `water_level_min`. Disse variablene er alle definert i `dam.h`.

3.9 D3: Water level restoration

Hvis vannstanden går utenfor de kritiske grensene for demningen må systemet iverksette tiltak for å gjenopprette vannstanden til det ideelle.

De fysiske dam-begrensningene er beskrevet i innledningen. Mengden vann som kan strømme inn eller ut av dammen per tidsenhet finnes i klassemedlemsvariablene
`outflow_capacity` og `inflow_capacity`. Dette betyr at det ofte tar tid før vannstanden kan komme helt tilbake til ideelt nivå igjen. Merk at hvis forskjellen mellom faktisk og ideelt vannivå er mindre enn innstrømnings- eller utstrømningskapasiteten kan du sette vannstanden direkte til ideelt nivå. Du kan bruke `water_level_delta`-funksjonen definert i oppgave D1 for å sjekke dette.

Implementer funksjonen `restore_water_level` som gjenoppretter vannstanden i demningen hvis den er over eller under det ideelle nivået.

Theme 3: File input/output

Oppgavene i denne filen handler om å lese og skrive fra og til tekstfiler i komma-separerte verdier (CSV) filformat. For å løse oppgavene må du endre filene `damsim.cpp` og `sensors.cpp`.

3.10 F1: CSV file output

Løsningen for denne oppgaven skal implementeres i filen `damsim.cpp`.

Skriv en funksjon som skriver simuleringsdata i komma-separerte verdier (CSV) filformat til filstrømmen `output_file`. Skriv linjene til filstrømmen på følgende format:

```
<dam name>,<current timestep>,<current water level>
```

Du kan få navnet på demningen og nåværende vannstand fra funksjonene `get_name` og `get_water_level` fra Dam-objektet-referanse parameter v. Se deklarasjonen av `Dam`-klassen i `dam.h` for mer detaljer. Gjeldende `timestep` sendes inn i parameteren `cur_timestep`.

3.11 F2: CSV file reading

Foreløpig er sensoravlesningene som brukes i simuleringen hardkodet i konstruktøren til klassen `SensorsFile` i `sensors.cpp`.

```
timestep,dam,inflow,outflow
10,dam02,0,40
10,dam01,40,0
20,dam01,300,0
20,dam02,0,300
25,dam01,300,0
25,dam02,0,300
...
```

Figur 4: Excerpt from the sensor data CSV file `sensor_data.csv`.

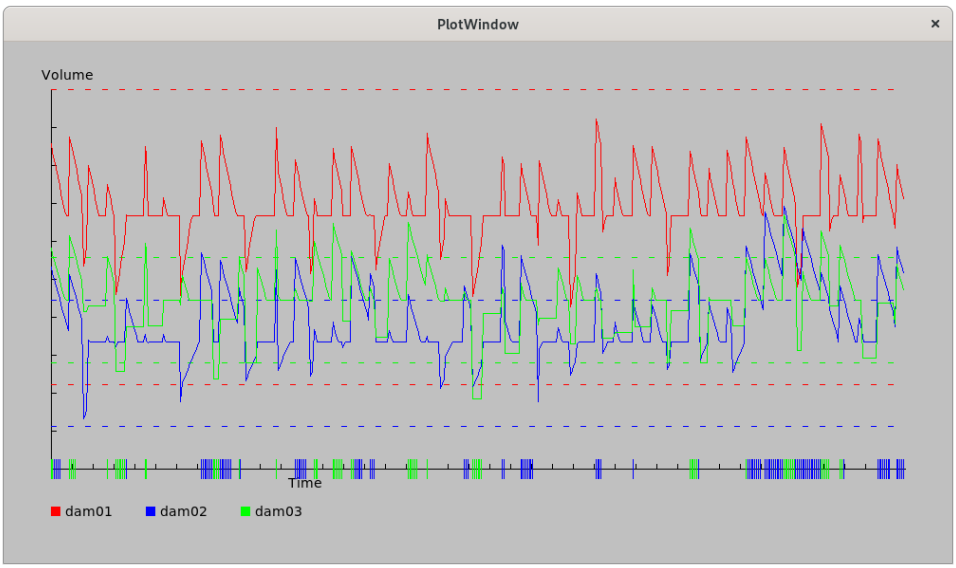
I denne oppgaven gjør vi dam-simulatoren mer fleksibel ved å utvide den med muligheten til å lese sensordata fra enkle CSV-filer[3]. Flere slike enkle CSV-filer følger med denne oppgaven i `data/`-mappen. Sensordatafilene heter `sensor_dataX.csv` der X er et heltall.

Figur 4 viser de første linjene i filen `sensor_data1.csv`. Som du kan se, er formatet som følger: Den første linjen i filen inneholder navnene på de komma-separerte kolonnene. I dette tilfellet er første kolonne `timestep`, den andre er `dam`-id, osv. De påfølgende linjene inneholder faktiske avlesninger. I dette eksemplet forekommer den første avlesningen på `timestep` 10 for dam02, med en `inflow` på 0 og `outflow` på 40.

Implementer konstruktøren til klassen `SensorsFile` i filen `sensors.cpp`. Funksjonen skal lese inn fra filnavnet som sendes som parameter `fname` til konstruktøren. For hver linje du leser inn, skal du kalle funksjonen `insert_reading` for å registrere avlesningen. Vi forventer ikke at du skal fokusere på unntakshåndtering som for eksempel feil i filformatet. Du kan derfor anta at funksjonen din bare vil bli brukt til å analysere filer som er helt i samsvar med formatet beskrevet ovenfor.

Siden de hardkodete standard sensoravlesningene tilsvarer målingene som finnes i `sensor_data1.csv`-filen, så bør du fremdeles se plottet som vises i Figur 3 hvis du implementerte funksjonen riktig. Hvis du ønsker, kan du prøve å plotte noen av de andre sensor-filene ved å endre variabelen `measurement_fil` i `main`-funksjonen i filen `main.cpp`.

[3]Merk at CSV-filene du møter i virkeligheten kan være mye mer komplisert enn disse.



Figur 5: An example of what the simulation plot with randomized sensor data might look like.

Theme 4: Class inheritance

Oppgavene i denne delen løses ved å endre filene `sensors.h` og `sensors.cpp`. I tillegg, for å teste implementasjonen din så må du endre template-parameteren i instansieringen av `DamSim` i `main.cpp`. Den relevante plassering i `main.cpp` er markert. Du må også huske å fjerne kommentar-tegnene på linjen nederst i `damsim.h` som anvist.

Default-implementasjonen som er delt ut bruker en forekomst av klassen `SensorFile` for å få sensoravlesninger. Denne filen implementerer den abstrakte klassen `Sensors` og leser sensordata fra en fil. I denne delen av eksamen ber vi deg om å implementere en sensorlesergenerator som genererer tilfeldige sensordata i stedet for å lese dem fra en fil.

En `Sensor` har bare en funksjon: `get_reading`. Funksjonen `get_reading` returnerer en `Reading struct` (definert i `sensors.h`) som inneholder målingene for oppgitt `Dam/timestep`. Hvis det ikke er noen sensorverdier å lese for en gitt `timestep og Dam`-kombinasjon, så skal funksjonen bare returnere en `Reading struct` med begge verdiene satt til 0.

Når du har implementert alle delspørsmålene i denne delen kan du ende opp med et plot som ligner på Figur 5. Resultatet ditt vil selvfølgelig avvike litt fra dette, avhengig av hvilken algoritme du velger for å generere tilfeldige avlesninger i `get_reading`-implementasjonen.

3.12 S1: Virtual class declaration

I `sensors.h`, skriv en klassedeklarasjon for `SensorsRandom`. Klassen skal inneholde deklarasjoner for funksjonene fra den abstrakte base-klassen `Sensors`. Implementer en delegerende konstruktør som kaller konstruktøren til den virtuelle klassen `Sensors`.

3.13 S2: Generating reading

Implementer funksjonen `get_reading` definert i `SensorsRandom`-klassen. Funksjonen får det gjeldende tidspunktet og navnet på dammen som parametere og returnerer en struct `Readings` (definert i `sensors.h`) som inneholder sensorverdiene.

Du står fritt til å bestemme logikken for denne funksjonen, den må imidlertid returnere fornuftige verdier for innstrømming og utstrømming generert ved hjelp av `rand ()`-funksjonen. For eksempel, en enkel tilnærming vil være å returnere tilfeldige verdier mellom 0 og 100 for innstrømming og utstrømmming hvert 10. timestep.

Slutten på denne eksamen.