



Norges teknisk–naturvitenskapelige
universitet
Institutt for datateknologi og
informatikk

TDT4102 Prosedyre-
og objektorientert
programmering
Vår 2024

Øving 7

Frist: 2024-03-08

Aktuelle temaer for denne øvingen:

- Klasser, arv, polymorfi og virtuelle funksjoner
- Tegning av former

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.
- 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:

- §14.2 og §14.3 i læreboken.

Merk: Siden vi bruker AnimationWindow istedenfor SimpleWindow, vil noen ting gjøres annerledes i boka. Se dokumentasjon til AnimationWindow [her](#).

1 Introduksjon til arv og polymorfi (15%)

a) Teorioppgave

Hva er forskjellen på `public`, `private` og `protected`?

Tips: les §14.3.4 i læreboka

b) Animal, en baseklasse

Du skal nå lage klassen `Animal`, som skal inneholde følgende medlemsvariabler:

- `name`, av typen `string`
- `age`, av typen `int`

Disse skal være `protected`. `Animal` skal ha en konstruktør som skal ta inn `name` og `age` som parametere, og initialisere medlemsvariablene. I tillegg skal den ha en `virtual` destruktør som frigjør minne. Destruktører lærer du om senere i emnet, så det eneste du trenger å vite nå er at de skrives slik:

```
virtual ~Animal() {}
```

Klassen skal også ha en `virtual` funksjon, `toString()`. som skal returnere: `"Animal: name, age"`

c) Cat og Dog, arvede klasser

Lag klassene `Cat` og `Dog`. De skal begge arve `public` fra `Animal`.

Begge klassene skal inneholde medlemsfunksjonen `toString()` som skal være `public`, og redefinere `toString()` i `Animal`-klassen. I `Cat`-klassen skal `toString()` returnere:

`"Cat: name, age"`. `toString()` i `Dog`-klassen skal returnere: `"Dog: name, age"`.

Begge klassene skal ha en konstruktør som kaller konstruktøren til `Animal`.

Nyttig å vite: `unique_ptr` (kort fortalt)

En peker (pointer) er en variabel som holder på minneadressen til et objekt. En enkel og trygg måte å opprette en peker på, er å benytte `std::unique_ptr`, som er en smart-peker.

Du kan lage en peker variabel som peker til et `Animal` objekt slik: `unique_ptr<Animal> dyrPtr`. For å få tilgang til medlemsfunksjonene til objektet som en peker "peker" på kan vi bruke `->` operatoren.

Pekere skal vi gå gjennom grundigere senere i kurset, dette er kun en liten intro slik at dere skal få til neste deloppgave.

d) Lag en testfunksjon, `testAnimal()`. Test klassene ved å opprette en `vector<unique_ptr<Animal>>` `animals`, og legg til noen instanser av hver klasse i denne. Vektoren holder pekere til `Animal` objekter, men vi kan også legge inn `Cat` og `Dog` objekter, siden disse også har `Animal` som datatype.

For å legge til en instans av typen `unique_ptr<Animal>` i vektoren, kaller du funksjonen `emplace_back()` slik: `animals.emplace_back(new object())`, hvor "object" erstattes med et kall til konstruktøren du har opprettet for klassen din, f.eks. `Cat{"Pus", 3}`.

Iterer gjennom vektoren og kall på `toString()` for hvert element. Siden vi bruker pekere gjøres dette slik: `animals.at(i) -> toString()`. Når vi kaller på `toString()` funksjonen skal vi få ulik funksjonalitet (`Animal:...`/`Cat:...`/`Dog:...`) ettersom hvilket objekt som kaller den. Dette er et eksempel på polymorfi.

Hva skjer hvis du fjerner `virtual` foran `toString()` i `Animal`-klassen?

e) Gjør `Animal`-klassen abstrakt. Dette kan du gjøre ved å endre `toString()` til å være en `pure virtual` funksjon. Hva skjer hvis du prøver å lage et `Animal` objekt nå?

2 Emoji (10%)

a) Emoji, en abstrakt baseklasse

I de utdelte filene ligger det skjelettkode som dere skal bruke videre i øvingen. **Denne oppgaven forklarer den utdelte koden. Du skal altså ikke skrive noe kode i denne oppgaven, men det er viktig at du forstår hvordan den utdelte koden er bygd opp, slik at du kan bruke den i de neste oppgavene.** De utdelte filene hentes via TDT4102-extensionsen som tidligere. Den utdelte filen `emoji_main.cpp` definerer en `main`funksjon. Pass på at du bare har en `main`funksjon i programmet ditt. Hvis du kompilerer og kjører skjelettkoden skal du få opp et tomt vindu.

I den utdelte koden er den abstrakte klassen `Emoji` definert. Det er viktig at du forstår hvorfor `Emoji` blir abstrakt og hvilke valg som er gjort for denne klassen før du tar fatt på resten av øvingen.

Det finnes mange forskjellige typer emoji: ansikter, hender, biler, båter, blomster, osv. Emoji i seg selv er et abstrakt konsept, vi kan ikke tegne konseptet "emoji", men vi kan snakke om konseptet og likevel forstå hva det innebærer. I denne øvingen skal vi modellere alle typer emoji med bakgrunn i at alle emoji har en felles operasjon. Denne operasjonen er ikke helt lik for alle emoji, så formålet er å gjøre det mulig å tegne en hvilken som helst emoji gjennom det samme grensesnittet. "Tegn smileansikt" eller "tegn bil" har til felles at operasjonen er "tegn". I vårt tilfelle er "tegn" noe som byttes ut med en bestemt funksjon som alle emoji-typer selv kan skrive over for å definere hvordan den spesifikke emoji skal tegnes.

Nesten alle `Emoji` har forskjellig antall og typer former: åpne og lukkede øyne, hår, strekmunn, smilemunn, øyebryn, ører, osv. Det gjør at alle de forskjellige emoji-klassene selv må ta ansvar for å tegne sine egne former til et vindu. Det er denne operasjonen vi bestemmer at alle emoji må ha, det felles grensesnittet.

`Emoji` har derfor en medlemsfunksjon som arvende klasser må overskrive for å bli konkrete, eller "følge kravet til grensesnittet". Medlemsfunksjonen har ansvar for å tegne de ulike øynene, munnene osv. til et `AnimationWindow`. Medlemsfunksjonen er *pure virtual* og heter `draw()`.

3 Ansikt (20%)

I denne oppgaven jobber vi med abstrakte klasser, og det blir derfor vanskelig å gjøre tester på dette stadiet. I oppgave 4 får vi derimot testet at alt fungerer som det skal, da vi skal lage konkrete klasser som arver fra disse abstrakte klassene. Dokumentasjon til [AnimationWindow](#) kan være nyttig i oppgavene videre.

a) Et utgangspunkt for ansikts-emoji.

Definer klassen `Face`, den skal arve fra `Emoji`.

Denne klassen skal representere et tomt ansikt, som skal være et utgangspunkt for ulike ansikts-emoji. [Emojipedia](#) holder en oversikt over hvilke smilefjes som finnes. Her finnes ingen smilefjes helt uten egenskaper. Derfor er det ikke ønskelig at smilefjes av typen `Face` skal kunne konstrueres.

Gjør derfor `Face` abstrakt. Det kan gjøres på samme måte som tidligere. `draw()` skal spesifiseres til å være *pure virtual* også for denne klassen.

b) **Face sine egenskaper.**

Selv om det ikke skal kunne instantieres objekter av typen **Face** har det en attraktiv egenskap som alle ansikter trenger. Nemlig et ansikt.

Klassen skal representere et sirkelformet ansikt og vi vil derfor at klassen skal inneholde medlemsvariablene **centre** og **radius** som er henholdsvis sirkelens posisjon og radius.

Klassen skal ha en konstruktør med to parametre, **Point c**, **int r**. Bruk medlemsinitialiseringsliste til å initialisere disse verdiene til medlemsvariablene **centre** og **radius**.

c) **Tegn ansiktet**

Overskriv **draw()** slik at det i vinduet tegnes en sirkel. Dette gjøres vha.

```
win.draw_circle(centre, radius, color);
```

color er fargen på sirkelen og er av typen **Color**, for eksempel **Color::yellow**.

Her bør du skrive **override** til slutt i deklarasjonen, slik at du får beskjed fra kompilatoren hvis du har skrivefeil e.l. i funksjonsnavn og parameterliste.

Selv om en funksjon er pure virtual kan den ha en definisjon. I dette tilfellet betyr pure virtual bare at klassen ikke kan brukes til å lage objekter.

4 Konkret emoji-klasse (20%)

a) **Ansikt med øyne, endelig en konkret klasse.**

Det er flere ansikter som har to åpne øyne som fellestrekk. Derfor skal du opprette en ansiktsklasse som har to øyne og arver fra den abstrakte klassen **Face**. Dette er en konkret emoji som kalles «empty face» eller «face without mouth». For å gjøre typenavnet litt ryddig skal denne klassen hete **EmptyFace**.

EmptyFace arver fra, og konstrueres på samme måte som **Face**, men skal i tillegg inneholde to øyne. Lag egne medlemsvariabler for posisjonen og radiusen til øynene.

Konstruktøren til **EmptyFace** skal gjenbruke konstruktøren til **Face**, for å initialisere ansiktet. Dette kalles å bruke *delegerende konstruktør*, og gjøres ved å kalle på konstruktøren til **Face** i initialiseringslisten.

```
EmptyFace(<parameterliste>) : Face{<argumenter>} /*, andre medlemmer */
```

Konstruktøren til **EmptyFace** skal også initialisere begge øynene. Størrelse på øyne og plassering av øyne i ansiktet bestemmer du selv. Plassering skjer ut fra ansiktets sentrum. Det er ikke et krav at øynenes plassering og størrelse skaleres i forhold til endringer i størrelsen til emoji. Det gjelder også resten av øvingen.

b) **Tegn EmptyFace i vinduet.**

draw() må overskrives for alle **Emoji**-deriverte klasser for at de skal bli konkrete og kunne tegnes i vinduet. Overskriv **draw()** funksjonen for **EmptyFace** klassen.

Former tegnes i den rekkefølgen de kalles. For at øynene skal vises må de tegnes til vinduet etter ansiktet. Ansiktet fra **Face** tegnes ikke automatisk til vinduet når **EmptyFace** overskriver **draw()**. Derfor må det eksplisitt kalles på **Face::draw()** for å tegne ansiktet.

EmptyFace::draw(AnimationWindow& win) bør derfor inneholde et kall til:

```
Face::draw(win);
```

c) **Tegn et tomt ansikt på skjermen.** Opprett et vindu og tegn ansiktet i vinduet. I den utdelte filen **emoji_main.cpp** er det allerede opprettet et vindu som kan brukes. Opprett et **EmptyFace** objekt og tegn det i vinduet ved å kalle på medlemsfunksjonen **draw()** med vinduet som parameter. *Dette er første gangen du kan teste koden din.* Juster programmet til du er fornøyd med plassering av øynene i ansiktet.

5 Flere emoji (35%)

Lag minst 5 forskjellige emojis. Du står fritt til å lage hvilke emoji du måtte ønske. a) til e) inneholder forslag til emojis du kan lage hvis du står helt fast. Gå inn på dokumentasjonen til [AnimationWindow](#) for å se hvordan ulike former tegnes.

Til slutt skal alle emojiene tegnes på skjermen. Du kan f.eks. lage en funksjon som tar inn `std::vector<std::unique_ptr<Emoji>& emojis` og et vindu emojiene skal tegnes til. Selv om alle emoji-klassene er forskjellige kan samme grensesnitt, `draw()`, brukes for å utrette samme operasjon på de forskjellige instansene av alle klassene som arver fra `Emoji`. Det er dette som gjør polymorfi ettertraktet.



Kunst til inspirasjon.

a) Smilefjes

Lag en smilefjesklasse, `SmilingFace`. Klassen skal arve fra `EmptyFace`, da er alt du trenger å gjøre å tegne en `arc` som representerer munnen.

b) Lei seg-fjes

Lag klassen `SadFace`. Du står fritt til å velge om du ønsker å arve fra smilefjeset i forrige deloppgave og justere på munnen fra det ansiktet, eller arve direkte fra `EmptyFace` og legge til en ny munn.

Hvorfor har du valgt det ene alternativet framfor det andre?

c) Sint ansikt

Lag klassen `AngryFace`. Se [emojipedias angry face](#) for inspirasjon.

d) Blunkeansikt

Lag klassen `WinkingFace`. Ansiktet skal ha ett åpent øye og ett øye som blinker. Det blinkende øyet kan f.eks. være to linjer som former en `<`-lignende form eller en halvsirkel. Se [emojipedias winking face](#) for inspirasjon.

e) Overrasket ansikt

Lag klassen `SurprisedFace`. Ansiktet du får når du skriver `:o` eller `:O`.

Du bestemmer selv om dette ansiktet skal arve fra et smilende ansikt med `arc`-munn som endres til å tegne 360 grader, eller om du benytter f.eks. `draw_circle()` til å representere munnen. Kan du identifisere potensielle problemer med å arve fra f.eks. `SmilingFace`?