Norges teknisk–naturvitenskapelige universitet
Institutt for datateknologi og informatikk

TDT4102 Prosedyre- og objektorientert programmering
Vår 2023

Øving 3

Frist: 2023-02-03

Mål for denne øvingen:

- Lære å organisere koden din i flere filer
- Lære å bruke nyttige funksjoner fra standardbiblioteket som `sin` og `cos` (Se PPP §24.8)
- Generere tilfeldige tall

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:

- Kapittel 5 og 8 i PPP.

Å skrive kode i flere filer

Når man programmerer er det viktig å være ryddig. C++ legger til rette for flere måter å organisere kode på. Å bygge en struktur ved hjelp av filer hjelper på lesbarhet og forståelse av koden. Uten denne muligheten ville alle programmer vi ønsket å skrive vært samlet i en stor fil. Du har kanskje merket at det kan bli mye skrolling, og vanskelig å finne det du leter etter når all koden er i en fil.

Vanligvis vil koden til et C++-program være strukturert som dette:

- En «hovedfil», gjerne kalt `main.cpp`, som inneholder `main`-funksjonen
- En eller flere `.cpp`-filer (implementasjonsfiler)
- En eller flere `.h`- eller `.hpp`-filer (headerfiler)

Hovedfilen

Dette er implementasjonsfilen som inneholder `main`-funksjonen, som kjøres når programmet startes. Ofte inneholder denne filen *kun* `main`-funksjonen, og ingen andre funksjoner.

Implementasjonsfiler

I disse filene implementeres funksjonene som brukes i programmet. Det er vanlig å plassere relaterte funksjoner i samme fil, gruppert etter formål. I dette faget kan det for eksempel være nyttig å ha én fil per oppgave i en øving, der man implementerer alle funksjonene som trengs for å gjøre den oppgaven.

Headerfiler

En *headerfil* er en fil med en samling av deklarasjoner. Headerfilen inneholder funksjonsprototyper for alle funksjoner som skal gjøres tilgjengelige når headerfilen inkluderes fra en implementasjonsfil. Å samle deklarasjoner i headerfiler er nøkkelen til god strukturering i C++. For å se hva et program kan gjøre er det nok å se i headerfilen, programmets grensesnitt, etter mulige operasjoner og handlinger.

Filenes relasjon til hverandre

En headerfil bør inkluderes i kildefiler som definerer funksjoner som er deklarert i headerfilen. Det gjør at kompilatoren kan luke ut eventuelle skrivefeil og andre småfeil som kan være vanskelig å oppdage. Det samme gjelder for kildefiler med kode som kaller på funksjonen, de må også kjenne til at funksjonen eksisterer og deklarasjonen må være tilgjengelig.

Konvensjonen er at hver `.cpp`-fil — med unntak av «hovedfilen» — har en headerfil med samme navn, som beskriver hva som er implementert i den tilsvarende `.cpp`-filen. Har man en fil som heter `myfunctions.cpp` har man også `myfunctions.h`. For å finne ut hva som er implementert i `myfunctions.cpp` kan man se på `myfunctions.h`.

Kopier og lim inn

De stedene det står `#include "headerfile.h"` er det enkelt og greit en innlesing av headerfilen som foregår ved kompilering. Se §8.3 *Headerfiles* i læreboka for et eksempel.

Generelt er det ønskelig at hver fil som inngår i prosjektet ditt bare blir med en gang i kompileringen. For headerfiler må man ofte passe på dette ved bruk av såkalte «header guards», men en enklere og

etterhvert svært utbredt måte å sørge for dette er å skrive `#pragma once` øverst i headerfilen. Det er ikke standard, men støttet av de aller fleste kompilatorer[1]

Under er et eksempel på hvordan et prosjekt som i tillegg til å ha en hovedfil har en implementasjonsfil og en headerfil kan se ut.

Headerfil(myfunctions.h)

```h
#pragma once
int add(int a, int b);
void printAdd(int a, int b);
```

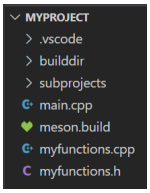
Implementasjonsfil(myfunctions.cpp)

```cpp
#include "myfunctions.h"
#include "std_lib_facilities.h"
// for å få tilgang på cout <<
int add(int a, int b){
    return a + b;
}
void printAdd(int a, int b){
    cout << add(a, b);
}
```

Hovedfil(main.cpp)

```cpp
#include "std_lib_facilities.h"
#include "myfunctions.h"
int main(){
    int a = add(3, 7);
    printAddtion(5, 6);
}
```

Figuren under viser hvordan filene er plassert i prosjektet.



For å kunne kjøre et prosjekt som inneholder flere `.cpp` filer er det viktig at man forteller `meson` om de andre filene, slik at ikke bare `main.cpp` blir bygd. Dette kan gjøres manuelt ved å gå inn i `meson.build` filen og legge de inn. Der det står `src = ['main.cpp']` må du legge til alle andre `.cpp` filer i listen.

Figuren under viser hvordan `meson.build` filen skal se ut for eksempelet over.

Alternativt kan man trykke `Ctrl+Shift+P` (`Cmd+Shift+P`) og kjøre *Configuration only* og deretter trykke overwrite. Da skal de nye filene dukke opp i listen i `meson.build`.

I denne øvingen kommer vi til å se på gjenstander i bevegelse. Spesifikt ønsker vi å se på banen til en kanonkule som blir skutt ut med en gitt vinkel og fart.

Opprett et prosjekt som beskrevet i øving 0.

[1] Du kan lese mer om dette her: https://en.wikipedia.org/wiki/Pragma_once.

1 Funksjonsdeklarasjoner (10%)

I denne oppgaven skal vi lage et sett med funksjonsdeklarasjoner. De skal være i en egen header-fil, som i denne øvingen skal hete `cannonball.h` (evt. `cannonball.hpp`). I denne oppgaven skal du kun *deklarere* funksjoner, ikke *definere* dem.

Funksjonsdeklarasjonene du skal lage i denne oppgaven danner grunnlaget for et *bibliotek* som kan regne ut banen til en kanonkule. Det skal ikke være nødvendig å gjøre antagelser.

a) Akselerasjonen i y-retning (oppover).
Funksjonen skal returnere akselerasjonen i y-retning og skal være et flyttall, (`double`). Funksjonen skal hete `acclY` og parameterlisten er tom.

b) Farten i y-retning (oppover).
Funksjonen tar inn to flyttall (`double`): startfart (`initVelocityY`) og tid i sekunder (`time`). Returverdien er farten som et flyttall. Funksjonen skal hete `velY`.

c) Posisjon i henholdsvis x- og y-retning.
Vi trenger en funksjon for hver retning:

Posisjon i x-retning
Posisjon i y-retning

Begge funksjonene tar inn tre flyttall hver: startposisjon (`initPosition`), startfart (`initVelocity`) og tid (`time`). Kall funksjonene `posX` og `posY`. Returtypen er også flyttall.

d) Utskrift av tid.
Den tar inn tid i sekunder (`double`) og returnerer ingen verdi. Funksjonen skal hete `printTime`.

e) Flyvetid.
Den tar inn startfart i y-retning og returnerer flyvetiden i sekunder. Funksjonen skal hete `flightTime`. Både parameter og returverdi er flyttall (`double`).

2 Implementer funksjoner (15%)

I denne oppgaven skal vi implementere funksjonene fra forrige oppgave. Alle funksjonsimplementasjoner skal ligge i en implementasjonsfil. I dette tilfellet bør den hete `cannonball.cpp` siden vi skal implementere funksjonene fra `cannonball.h`.

De nødvendige formlene vil bli presentert. Din oppgave er å skrive funksjoner som passer til beskrivelsen/formelen under hver oppgave og som samsvarer med funksjonsdeklarasjonene vi lagde tidligere. For å få hjelp av kompilatoren til å unngå skrivefeil er det en god tommelfingerregel å inkludere headerfilen(e) som inneholder deklarasjoner for de funksjonene som skal implementeres. F.eks.

```
#include "cannonball.h"
```

Legg header- og implementasjonsfil i samme filkatalog, så unngår du unødvendige problemer.

a) Akselerasjon i y-retning.
Vi definerer vertikal akselerasjonen til å ha positiv retning oppover, derfor blir kulas akselerasjon $-9.81\,\mathrm{m\,s^{-2}}$.

b) Fart i y-retning.
Funksjonen skal beregne farten i y-retningen basert på argumentene:

$$\text{FartY} = \text{StartFartY} + \text{AkselerasjonY} \cdot \text{Tid} \tag{1}$$

c) Posisjonsberegning.

Formel for beregninger (gjelder både x- og y-retning):

$$\text{Posisjon} = \text{StartPosisjon} + \text{StartFart} \cdot \text{Tid} + \frac{\text{Akselerasjon} \cdot \text{Tid}^2}{2} \tag{2}$$

Akselerasjon i x-retningen er $0\,\mathrm{m\,s^{-2}}$

d) Utskrift av tid.

Funksjonen skal fra argumentet beregne antall timer, minutter og sekunder og skrive det til skjerm på en leselig måte.

e) Flyvetid.

`flightTime` skal finne ut hvor lenge et objekt kommer til å fly gitt en startfart. I denne oppgaven antas et plant underlag, ingen luftmotstand, og kun tyngdekraften påvirker kulens fart i y-retningen, slik at kun startfarten i y-retning påvirker flytiden. Vi antar også at kulen skytes ut fra (posY = 0) og får dermed at kulens flytid er gitt ved:

$$\text{Tid} = \frac{-2 \cdot \text{StartFartY}}{\text{AkselerasjonY}} \tag{3}$$

3 Verifiser at funksjonene fungerer (15%)

Å teste et program er viktig for å verifisere at oppførselen er som forventet. Det kan være flere kilder til en feil: syntaks, logikk, eksterne faktorer, osv. Ofte kan det være utfordrende å finne feil dersom man skriver mye kode før den testes. Vi oppfordrer sterkt til å teste koden underveis mens du jobber med øvingene, selv om det ikke er en eksplisitt oppgave.

a) Forsikre deg om at programmet kompilerer.

Dersom programmet ikke kompilerer, sjekk følgende:

- Filer eksisterer: `main.cpp`, `cannonball.h` og `cannonball.cpp`.
- Inkludering: `main.cpp` skal inkludere `cannonball.h`.
  `cannonball.cpp` bør inkludere `cannonball.h`.
- `meson.build` filen inneholder `src = ['main.cpp', 'cannonball.cpp']`.
- I alle filene finnes det til sammen kun en main-funksjon (`int main()`).
- .cpp-filer bør aldri inkluderes, kun .h-filer.
- Funksjonene har samme navn, returverdi og parameterliste, når den deklareres og når den defineres
- Ingen funksjoner er definert i header-filer
- Alle funksjoner er definert kun en gang

Dersom det fortsatt ikke fungerer når du prøver å kompilere må du se nærmere på feilmeldingene. Forstår du ikke feilmeldingen kan du prøve å søke etter den med Google, spørre studentassistenten din eller spørre i emnets diskusjonsforum.

Nyttig å vite: flyttall

Når man gjør operasjoner med flyttall kan man få et annet resultat enn det man forventer. Dette er fordi en datamaskin representerer flyttall med et endelig antall desimaler. Du kan lese mer om dette i lærebokas kapittel 24.2. Når vi skal verifisere at funksjon som utfører flyttalloperasjoner gir oss ønsket resultat må vi sjekke om resultatet ligger innenfor et intervall. Altså at resultatet er tilnærmet likt forventet resultat.

Ved å bruke eksempeldata der man vet svaret på forhånd kan man teste at programmet fungerer som ønsket.

Tabellen under viser forventede verdier for `velX`, `velY`, `posX` og `posY` ved tidspunktene $t = 2.5\,s$ og $t = 5\,s$. Startposisjonen i x- og y-retning er 0 når $t = 0\,s$. Og startfarten i x-retning er $50\,m\,s^{-1}$ og startfarten i y- retning $25\,m\,s^{-1}$ når $t = 0\,s$.

	t = 2.5	t = 5.0
velX	50.0	50.0
velY	0.475	-24.05
posX	125.0	250.0
posY	31.84	2.375

b) Lag en testfunksjon.

Opprett en funksjon som skal sammenligne to flyttall og avgjøre om de er tilnærmet like. Funksjonen skal skrive resultatet av testen til skjerm. Deklarasjonen til funksjonen skal være:

```
void testDeviation(double compareOperand, double toOperand,
                   double maxError, string name)
```

og funksjonen skal teste om |compareOperand-toOperand| ≤ maxError holder.

Deklarer og implementer funksjonen i `main`-filen.

Nyttig å vite: deklarasjon og definisjon i samme fil

En funksjon kan deklareres et sted og defineres et annet sted, også i samme fil, for eksempel i main-filen. Da kan funksjonen deklareres over `main()` og defineres under. På den måten vet `main()` at funksjonen finnes uten at definisjonen må ligge før.

```
int add(int a, int b); // deklarasjon
int main() {
cout << add(2, 3) << endl;
}
int add(int a, int b) { return a + b; } // definisjon
```

c) Test hver funksjon fra main.

Test funksjonene dine med `testDeviation()`. Et eksempel på bruk av funksjonen er

```
testDeviation(posX(0.0,50.0,5.0), 250.0, maxError, "posX(0.0,50.0,5.0)");
```

Her tester vi om funksjonen `posX()` gir riktig resultat. Her er `maxError` en konstant i programmet ditt som du definerer verdien på selv (f.eks. 0.0001), og vi bruker parameteren `name` til å si hvilken test vi utfører. Om funksjonen `posX()` gir riktig resultat vil `testDeviation()` gi utskrift "posX(0.0,50.0,5.0) is valid." eller lignede. Om den gir et resultat utenfor feilmarginen vil `testDeviation()` gi utskrift "posX(0.0,50.0,5.0) is not valid, expected 250.0, got 125.0." eller lignende.

4 Implementer funksjoner (20%)

a) Implementering av funksjoner.

Implementer funksjonene under i `cannonball.cpp`. Husk å plassere deklarasjonene i `cannonball.h`

```
// Ber brukeren om en vinkel
double getUserInputTheta();

//Ber brukeren om en absoluttfart
double getUserInputAbsVelocity();

// Konverterer fra grader til radianer
double degToRad(double deg);

// Returnerer henholdsvis farten i x-, og y-retning, gitt en vinkel
// theta og en absoluttfart absVelocity.
double getVelocityX(double theta, double absVelocity);
double getVelocityY(double theta, double absVelocity);

// Dekomponerer farten gitt av absVelocity, i x- og y-komponentene
// gitt vinkelen theta. Det første elementet i vectoren skal være
//x-komponenten, og det andre elementet skal være y-komponenten.
// "Vector" i funksjonsnavnet er vektor-begrepet i geometri
vector<double> getVelocityVector(double theta, double absVelocity);
```

Funksjonene brukes til å lese inn en vinkel og en fart fra brukeren og omgjøre de innleste verdiene til hastighet i x- og y-retning. Den siste funksjonen, `getVelocityVector`, kombinerer så de to forrige getVelocity-funksjonene i én funksjon.

I funksjonen `getVelocityX` beregnes farten i x-retning:

$$\text{FartX} = \text{AbsoluttFart} \cdot \cos(\text{Vinkel}) \tag{4}$$

Tilsvarende for `getVelocityY`:

$$\text{FartY} = \text{AbsoluttFart} \cdot \sin(\text{Vinkel}) \tag{5}$$

Funksjonen `getVelocityVector()` kaller `getVelocityX()` og `getVelocityY`() og lagrer returverdiene fra disse funksjonene i en `vector` og returnerer denne.

Om man kjører `getVelocityVector()` med en vinkel på 27.5 grader og absolutt hastighet på 15.64 skal vektoren med resultatene være $\approx \{13.8728, 7.22175\}$.

b) Implementer funksjonen

```
double getDistanceTraveled(double velocityX, double velocityY)
```

som skal returnere den horisontale avstanden kanonkulen fløy før den traff bakken, med andre ord verdien til posisjonen i x-retning når posisjonen i y-retning (høyde over bakken) er 0. Gjenbruk av kode anbefales.

Du kan fremdeles anta at kanonkulen skytes ut fra $posY = 0$.

Ved bruk av resultatene fra oppgave a) skal du få en distanse på 20.4253.

c) Implementer funksjonen `targetPractice()`

som skal ta inn en avstand `distanceToTarget` og returnere avviket i meter mellom verdien `distanceToTarget` og der kulen lander (avstand fra start i x-retning) når `velocityX` og `velocityY` er henholdsvis startfart i x- og y-retning.

```
double targetPractice(double distanceToTarget,
                      double velocityX,
                      double velocityY);
```

d) Test koden fra `main()`.

Verifiser at programmet oppfører seg som du forventer.

e) Advarsler

Funksjonen under er skrevet for å finne ut om man har truffet blink. Koden er imidlertid dårlig, og vil gi en advarsel (*eng: warning*) når den bygges. Kopier koden, kall den i `main()` og kompiler. Hvilken advarsel får du i terminalen?

Bytt så ut parameterne i `targetPractise()` til for eksempel `targetPractice(100,0,0)`. Hva skjer nå? Hvorfor skjer dette?

```
bool checkIfDistanceToTargetIsCorrect() {
    double error = targetPractice(0,0,0);
    if(error == 0) return true;
}
```

Noen typer advarsler er ikke alvorlige, mens andre, som eksempelet over, kan få programmet til å feile eller gi uforventet oppførsel hvis de ikke tas hånd om. Man bør derfor rette opp i advarslene som dukker opp, selv om koden bygger og kjører.

5 Cannonball, et lite spill (40%)

I denne oppgaven skal du lage et enkelt spill som går ut på å skyte en kanonkule mot et mål.

Ofte har vi lyst til at programmet vårt skal ha en form for tilfeldig oppførsel. Vanligvis får man til dette ved å generere *tilfeldige tall*. Når vi her snakker om tilfeldige tall mener vi *pseudotilfeldige* heltall fra en *uniform sannsynlighetsfordeling* på et bestemt *intervall*, som vil si at alle de mulige verdiene i intervallet skal være like sannsynlige utfall. At vi genererer *pseudotilfeldige* tall betyr at tallene regnes ut med en deterministisk algoritme basert på en «startverdi» som kalles et *frø* (Eng. *seed*). Datamaskinger (og mennesker) er dårlige på å generere faktisk tilfeldige tall.

Nyttig å vite: <random>

Det er flere måter å finne tilfeldige verdier i C++, en av disse er ved bruk av <random> headeren.

`std::default_random_engine` er en av klassene som benyttes. Denne klassen har et innebygd intervall den tar pseudotilfeldige verdier fra, og kan brukes på denne måten:

```
int main()
{
    std::default_random_engine generator;
    int number = generator();

    std::cout << number;
}
```

Dersom man kjører koden over flere ganger, får vi samme resultat hver gang, dette kan fikses ved å gi `std::default_random_engine` et seed. For at vi skal få ulike seed hver gang tar vi i bruk `std::random_device`. Ved bruk av dette kan vi lage tilfeldige tall på følgende måte:

```
int main()
{
    std::random_device rd;
    std::default_random_engine generator(rd());

    int number = generator();

    std::cout << number;
}
```

Ved å kjøre kodesnutten over vil en få et tall av en stor størrelsesorden, for å få verdier på intervallet man vil, med fordelingen man ønsker kan man ta i bruk en distribution-funksjon fra <random>, for å få en jevn fordeling over intervallet man ønsker kan man bruke enten `std::uniform_int_distribution<int>` eller `std::uniform_real_distribution<double>`. Med `<type>` menes datatypen man ønsker, for eksempel `int` eller `double`.
`std::uniform_int_distribution<type>` brukes når man vil ha integers eller andre tilsvarende datatyper, om en eksempelvis skal ha en tilfeldig char fra `'A'` til og med `'Z'` skrives `std::uniform_int_distribution<char> distribution('A', 'Z')`.
`std::uniform_real_distribution<type>` brukes når man vil ha double eller andre tilsvarende datatyper. Om man vil ha ti flyttal i intervallet [0,1) kan man gjøre det på følgende måte:

```
int main()
{
    std::random_device rd;
    std::default_random_engine generator(rd());
    std::uniform_real_distribution<double> distribution(0,1);

    for (int i = 0; i<10; i++){
        double number = distribution(generator);
        std::cout << number << '\n';
    }
}
```

a) Opprett en ny fil, `utilities.cpp`, og tilhørende headerfil.
Filen skal inneholde en funksjon for å generere pseudotilfeldige tall.

b) Skriv en funksjon `randomWithLimits` som tar inn en øvre og nedre grense, og som bruker `default_random_engine` til å returnere et heltall i det lukkede intervallet gitt av disse grensene. Forsikre deg om at denne funksjonen fungerer slik den skal ved å teste den i `main`-funksjonen. Bruk en løkke for å kjøre funksjonen flere ganger. Kjør programmet flere ganger og sammenlign kjøringene.

c) I forrige oppgave fikk vi samme resultat ved hver kjøring. Endre programmet ditt slik at du får **forskjellig resultat ved hver kjøring, ved å bruke `random_device`.** Test deretter programmet ditt igjen, som beskrevet i forrige deloppgave. *Hint: Bruk `random_device` som beskrevet over.*

d) Implementer funksjonen `playTargetPractice()` i `cannonball.cpp`

```
void playTargetPractice();
```

I denne funksjonen skal du lage et enkelt spill. Spillet går ut på å skyte en kanonkule mot en blink. Spilleren vinner hvis hun treffer blinken.

Blinken skal plasseres tilfeldig mellom 100 og 1000 meter fra kanonen (blinken skal kun plasseres en gang). Spilleren får ti forsøk på å treffe blinken. Spilleren skyter kulen ved å oppgi inn en vinkel og en startfart. For hvert forsøk skal avstanden til målet, samt om skuddet var for langt eller for kort, skrives til skjerm. I tillegg skal hvor lang tid kulen har brukt på reisen skrives til skjerm, på en måte som er lett å lese for mennesker.

Dersom kanonkulen lander mindre enn fem meter unna målet regnes det som et treff, og spilleren har vunnet. Spilleren taper dersom hun ikke treffer på ti forsøk. Hvis spilleren vinner skal hun gratuleres, og hvis hun taper skal hun trøstes.

e) Visualisering av kanonkulen — frivillig —

For å hente koden som inneholder visualiseringsfunksjonen gjør det følgende:

1. Trykk `ctrl+shift+P`(`cmd+shift+P` på mac) for å åpne «Command Palette» i vscode.
2. Begynn å skriv og velg «TDT4102: Create Project From TDT4102 Template»
3. Velg mappen «Exercises» og deretter «O03»

Du skal nå ha fått to filer i prosjektet ditt: `cannonball_viz.h` og `cannonball_viz.cpp`.

Dersom Exercises-mappen ikke dukker opp, kjør «TDT4102: Look for update to the course content» fra «Command Palette».

I disse filene er det skrevet kode for å visualisere kanonkulens bane. Kall funksjonen

```
void cannonBallViz(double targetPosition, int fieldLength,
                   double velocityX, double velocityY, int timeSteps)
```

for å åpne et vindu som animerer kulens bane. `targetPosition` er blinkens posisjon, `fieldLength` skal settes til den maksimale avstanden (1000 meter) og `timeSteps` (tidssteg) bestemmer hvor mange ganger langs kulens bane man skal vise kulens posisjon. Kallet på funksjonen skal gjøres en gang inne i den løkka der du gir tekstlig feedback til spilleren.