

Norges teknisk-naturvitenskapelige  
universitet  
Institutt for datateknologi og  
informatikk

TDT4102 Prosedyre-  
og objektorientert  
programmering  
Vår 2025

Øving 3

**Innleveringsfrist: 2025-01-31**

**Mål for denne øvingen:**

- Lære å organisere koden din i flere filer
- Lære å bruke nyttige funksjoner fra standardbiblioteket som `sin` og `cos`
- Generere tilfeldige tall

**Generelle krav:**

- Bruk de eksakte navn og spesifikasjoner gitt i oppgaven.
- 70% av øvingen må godkjennes for at den skal vurderes som bestått.
- Øvingen skal leveres på **INGInious**.
- Benytt VS Code til å skrive, kompilere og kjøre kode.

**Anbefalt lesestoff:**

- Kapittel 4 og 7 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 deklart 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 §7.7 *Modules and headers* i læreboka for et eksempel.

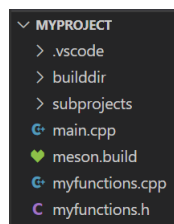
Generelt er det ønskelig at hver fil som inngår i prosjektet ditt bare blir med en gang i kompileringen. For headerfiler må man ofte passe på dette ved bruk av såkalte «header guards», men en enklere og

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<sup>1</sup>

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.

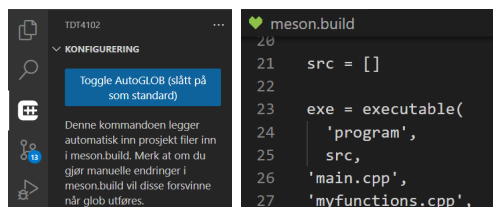
<b>Headerfil(myfunctions.h)</b> <pre>#pragma once int add(int a, int b); void printAdd(int a, int b);</pre>	<b>Implementasjonsfil(myfunctions.cpp)</b> <pre>#include "myfunctions.h" #include "std_lib_facilities.h" // for å få tilgang på cout &lt;&lt; int add(int a, int b){     return a + b; } void printAdd(int a, int b){     cout &lt;&lt; add(a, b); }</pre>
<b>Hovedfil(main.cpp)</b> <pre>#include "std_lib_facilities.h" #include "myfunctions.h" int main(){     int a = add(3, 7);     printAdd(5, 6);     return 0; }</pre>	

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 kompilatoren om de andre filene, slik at ikke bare `main.cpp` blir bygd. For at kompilatoren skal kunne kompilere alle filene i prosjektet ditt, må filnavnene ligge i `meson.build`-filen. Dette gjøres automatisk av VS Code-utvidelsen til faget, men i tilfelle man skulle ønske å gjør det manuelt, er det laget en egen Toggle AutoGLOB-knapp som skrur av denne funksjonen. Den ligger øverst når du trykker inn på C++ ikonet i venstre kolonner i VS Code. GLOB-funksjonen kan gjøres manuelt ved å gå inn i `meson.build`-filen og legge inn alle `.cpp`-filene i prosjektet ditt.

Figurene under viser hvordan Glob-knappen ser ut og hvordan `meson.build` filen skal se ut for eksempelet over etter at `'myfunctions.cpp'` er lagt inn.



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.

<sup>1</sup>Du kan lese mer om dette her: [https://en.wikipedia.org/wiki/Pragma\\_once](https://en.wikipedia.org/wiki/Pragma_once).

Opprett et prosjekt som beskrevet i øving 0 med templatene "Exercises/O03". Åpne kommando-paletten med **Ctrl+Shift+P** (**Cmd+Shift+P** om du er på Mac) → **Create project from template** → **Exercises** → **003**.

For hver deloppgave vil man finne to kommentarer som definerer henholdsvis begynnelsen og slutten av svaret ditt. Kommentarene er på formatet: `// BEGIN: <oppgavenummer><deloppgave> og // END: <oppgavenummer><deloppgave>`.

I oppgave 1a skriver man mellom `// BEGIN: 1a` og `// END: 1a`. I oppgave 1b skriver man mellom `// BEGIN: 1b` og `// END: 1b` osv. Kun kode som blir skrevet innenfor kommentarene rettes. Det er viktig at ikke **BEGIN**- eller **END**-kommentarene fjernes da dette fører til at oppgaven ikke blir automatisk rettet og 0 poeng blir gitt.

## 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 heter `cannonball.h`. I denne oppgaven skal du kun *deklare* funksjoner, ikke *definere* dem.

Funksjonsdeklarasjonene du skal lage i denne oppgaven danner grunnlaget for et *bibliotek* som kan regne ut banen til en kanonkule.

**NB:** Her er det ekstremt viktig at funksjonsnavn og parameterlisten er nøyaktig lik som gitt i oppgaven.

**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`). Returner verdien 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 (`initPositionX/Y`) og startfart (`initVelocityX/Y`) i den respektive retningen, og tid (`time`). Kall funksjonene `posX` og `posY`. Returnertypen 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 (`initVelocityY`) 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 implementasjonsfilen. I dette tilfellet heter den `cannonball.cpp` siden vi skal implementere funksjonene fra `cannonball.h`.

Din oppgave er å skrive funksjoner som passer til beskrivelsen/formelen under hver oppgave og som samsvarer med funksjonsdeklarasjonene i `cannonball.h`. 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. I `cannonball.cpp` ser det slik ut:

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

### a) Akselerasjon i y-retning.

Vi definerer vertikal akselerasjonen til å ha positiv retning oppover, derfor blir kulas akselerasjon  $-9.81 \text{ m s}^{-2}$ .

### b) Fart i y-retning.

Funksjonen skal beregne farten i y-retningen basert på argumentene:

$$\text{FartY} = \text{StartFartY} + \text{AkselerasjonY} \cdot \text{Tid} \quad (1)$$

### c) Posisjonsberegning.

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

$$\text{Posisjon} = \text{StartPosisjonX/Y} + \text{StartFartX/Y} \cdot \text{Tid} + \frac{\text{AkselerasjonX/Y} \cdot \text{Tid}^2}{2} \quad (2)$$

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

### 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  $\text{posY} = 0$  og får dermed at kulens flytid er gitt ved:

$$\text{Tid} = \frac{-2 \cdot \text{StartFartY}}{\text{AkselerasjonY}} \quad (3)$$

### 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 `'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 deklarerer 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(e) kan du prøve å søke etter den med din favorittsøkemotor, spørre studentassistenter på sal, spørre ChatGPT, eller spørre på Piazza.

#### 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. 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.

#### b) Lag en testfunksjon.

Opprett en funksjon som skal sammenligne to flyttall og avgjøre om de er tilnærmet like. Funksjonen skal teste om uttrykket  $|\text{compareOperand-toOperand}| \leq \text{maxError}$  holder og skrive resultatet av testen til skjerm. Den skal i tillegg returnere `true` dersom testen holdt og `false` ellers. Funksjonen er allerede deklart i `cannonball.h` og skal implementeres i `cannonball.cpp`.

#### Nyttig å vite: deklarasjon og definisjon i samme fil

En funksjon kan deklarerer et sted og defineres et annet sted, også i samme fil, for eksempel i `main`-filen. Da kan funksjonen deklarerer 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
```

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  $50 \text{ m s}^{-1}$  og startfarten i y- retning  $25 \text{ m s}^{-1}$  når  $t = 0$  s.

	$t = 2.5$	$t = 5.0$
<code>velX</code>	50.0	50.0
<code>velY</code>	0.475	-24.05
<code>posX</code>	125.0	250.0
<code>posY</code>	31.84	2.375

## 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 lignende. 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 og deklarer funksjoner (20%)

## a) Implementering og deklarering av funksjoner.

Implementer funksjonene under i `cannonball.cpp`. Husk å plassere deklarasjonene i `cannonball.h`. Du finner noen nyttige formler lengre ned i oppgaven. **NB:** Her er det ekstremt viktig at funksjonsnavn og parameterlisten er nøyaktig lik som gitt i oppgaven.

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

//Ber brukeren om en startfart
double getUserInputInitVelocity();

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

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

// Dekkomponerer farten gitt av initVelocity, i x- og y-komponentene
// gitt vinkelen theta. Det første elementet i vektoren 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 initVelocity);
```

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:

$$\text{FartX} = \text{StartFart} \cdot \cos(\text{VinkelRadianer}) \quad (4)$$

Tilsvarende for `getVelocityY`:

$$\text{FartY} = \text{StartFart} \cdot \sin(\text{VinkelRadianer}) \quad (5)$$

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 start hastighet på 15.64 skal vektoren med resultatene være  $\approx \{13.8728, 7.22175\}$ .

## b) Implementer og deklarer 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. *Hint: funksjonene `flightTime` og `posX` kan være nyttige*

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 og deklarer 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.



## 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*). Datamaskiner (og mennesker) er dårlige på å generere faktisk tilfeldige tall.

### Nyttig å vite: <random>

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

Vi skal benytte klassen `default_random_engine` som ligger i dette biblioteket. Denne klassen har et innebygd intervall som den tar pseudotilfeldige verdier fra, og kan brukes på denne måten:

```
int main() {
    default_random_engine generator;
    int number = generator();
    cout << number;
    return 0;
}
```

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

```
int main() {
    random_device rd; //seed
    default_random_engine generator(rd());
    int number = generator();
    cout << number;
    return 0;
}
```

Ved å kjøre kodesnutten over vil en få et tall mellom 1 og et veldig stort tall. For å få verdier på intervallet man vil, med den fordelingen man ønsker, kan man ta i bruk en distribusjon-funksjon fra <random>-biblioteket. For å få en jevn fordeling over intervallet man ønsker kan man bruke enten `uniform_int_distribution<type>` eller `uniform_real_distribution<type>`. Med `type` menes datatypen man ønsker, for eksempel `int` eller `double`.

`uniform_int_distribution<type>` brukes når man vil ha `int` eller andre tilsvarende datatyper. `uniform_real_distribution<type>` brukes når man vil ha `double` eller andre tilsvarende datatyper.

Om man vil generere ti flyttal i intervallet [0,1) kan man gjøre det på følgende måte:

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

    for (int i = 0; i<10; i++){
        double number = distribution(generator);
        cout << number << endl;
    }
    return 0;
}
```

- a) **Skriv en funksjon `randomWithLimits`** som tar inn en nedre og øvre grense som begge har datatype `int`, og returnere et pseudotilfeldig heltall (`int`) i det lukkede intervallet gitt av disse grensene. Funksjonen skal deklarerer i `utilities.h` og implementeres i `utilities.cpp`.

Forsikre deg om at denne funksjonen fungerer slik den skal ved å teste den i `main`-funksjonen. Generer flere tilfeldige tall ved hjelp av en løkke. Kjør programmet flere ganger og se at du får forskjellige tall hver kjøring.

- b) Implementer funksjonen `playTargetPractice()` i `cannonball.cpp` og deklarerer den i `cannonball.h`.

```
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 hen 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 kula ved å oppgi inn en vinkel og en startfart. For hvert forsøk skal avstanden skuddet var unna målet, samt om skuddet var for langt eller for kort, skrives til skjerm. I tillegg skal hvor lang tid kula har brukt på reisen skrives til skjerm, på en måte som er lett å lese for mennesker.

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

- c) **Visualisering av kanonkula — frivillig —**

Ta en titt i filene: `cannonball_viz.h` og `cannonball_viz.cpp`.

I disse filene er det skrevet kode for å visualisere kanonkulens bane. Koden er kommentert ut i `cannonball_viz.cpp` for å unngå kompileringsfeil tidligere i øvingen, men nå kan du fjerne `/* */` for å bruke den utgitte koden.

Kall funksjonen

```
void cannonBallViz(double targetPosition, int fieldLength,
                  double initVelocityX, double initVelocityY, 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.