



Norges teknisk-naturvitenskapelige  
universitet  
Institutt for datateknologi og  
informatikk

TDT4102 Prosedyre-  
og objektorientert  
programmering  
Vår 2019

Øving 3

**Frist: 2019-02-01**

### Mål for denne øvingen:

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

### Generelle krav og anbefalinger:

- Bruk de eksakte navn og spesifikasjoner gitt i oppgaven.
- Teorioppgaver besvares med kommentarer i kildekoden slik at læringsassistenten enkelt finner svaret ved godkjenning.
- Det anbefales å benytte en programmeringsomgivelse (IDE) slik som Visual Studio eller XCode.
- 70% av øvingen må godkjennes for at den skal vurderes som bestått.
- Øvingen skal godkjennes av stud.ass. på sal.

### 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.

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)

### 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.

### Hovedfilen

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

### 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 `minefunksjoner.cpp` har man også `minefunksjoner.h`. For å finne ut hva som er implementert i `minefunksjoner.cpp` kan man se på `minefunksjoner.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 (PPP) 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>.

---

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

## Oppgaveforklaring:

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 om inkluderer `std_lib_facilities.h` for å gjennomføre den obligatoriske delen av denne øvingen. Siste deloppgave er frivillig og krever at du oppretter et grafikkprosjekt.

## Del 1: Enkel bevegelse

### 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`).

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 `acc1Y` 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  $-9.81 \text{ m s}^{-2}$ .

**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):

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

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 friksjon i lufta, 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 deler av 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.

**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`.
- I alle filene finnes det til sammen kun en main-funksjon (`int main()`).
- `.cpp`-filer bør aldri inkluderes, kun `.h`-filer.

Dersom det fortsatt ikke fungerer når du prøver å compilere 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.

**b) Test hver funksjon fra main.**

- Ved å bruke eksempeldata der man vet svaret på forhånd blir det enklere å finne feil i oppførselen til programmet. Dataene under er noen eksempler på forventede resultater, verifiser at det stemmer overens med ditt programs oppførsel.

	T = 0	T = 2.5	T = 5.0
acclX	0	0	0
acclY	-9.81	-9.81	-9.81
velX	50.0	50.0	50.0
velY	25.0	0.475	-24.05
posX	0.0	125.0	<b>250.0</b>
posY	0.0	31.84	2.375

*Merk: Hverken `acclX` eller `velX` tilsvarer funksjoner i oppgaven. Disse verdiene er med fordi du kanskje trenger dem som argumenter når du skal teste andre funksjoner. Vi antar at `initPosition = 0` i både x- og y-retning.*

- Som dere skal ha lært i ITGK så har flyttall begrenset presisjon, og derfor vil sammenlikning med likhetsoperatoren sjelden gi ønsket resultat<sup>2</sup>. Opprett en funksjon som kan sammenligne to flyttall og avgjøre om de er tilnærmet like. (Hint: ta absoluttverdien av differensen mellom de to tallene, og sjekk deretter om den er mindre enn en feilmarging (lite tall)). Benytt funksjonen til å avgjøre om de definerte funksjonene samsvarer med forventede resultat fra tabellen over. Deklarasjonen til en slik funksjon kan være `void testDeviation(double compareOperand, double toOperand, double maxError, string name)`. Et eksempel på bruk for den ene tallverdien gitt med fete typer i tabellen ovenfor (250.0) er:

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

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. Hint: funksjonen `abs(number)` for å ta absoluttverdi kan være nyttig her.

- En funksjon kan deklarerer et sted og defineres et annet, også i samme fil. Det kan være nyttig hvis `testDeviation()` defineres i `main.cpp`. Da kan funksjonen deklarerer over `main()` og defineres under. På den måten vet `main()` at funksjonen finnes uten at definisjonen må ligge før. Eks på `main.cpp`:

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

## Del 2: Gjenbruk av funksjoner

### 4 Implementer funksjoner (20%)

#### Nyttig å vite: referanser kan brukes for å få flere svar ut av en funksjon

I C++ kan vi i utgangspunktet kun returnere én verdi fra en gitt funksjon. For å omgå denne begrensningen kan bl.a. referanser benyttes. En referanse er et annet navn for et allerede eksisterende objekt - et alias. Se §8.5.4 til 8.5.6 i læreboka (PPP) for illustrasjoner og argumentasjon for når verdi og (const) referanse bør benyttes.

Et eksempel på en funksjonsdeklarasjon til en funksjon som tar inn en referanse som argument kan sees i `getUserInput()` nedenfor. `double&` betyr her at funksjonen tar inn en referanse til en `double`. Funksjonen benytter ellers helt lik syntaks som for en vanlig copy-by-value parameter når det gjelder å lese eller skrive til parameteren. Funksjonen får ingen kopi av verdien, men en referanse til objektet som gis som argument.

#### a) Implementering av funksjoner.

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

<sup>2</sup>Du kan lese mer om flyttall og presisjon i lærebokas kapittel 24.2

```

// Ber brukeren om to tall, en vinkel (i grader) og en fart.
// Formålet er å endre verdien på argumentene.
// theta og absVelocity er referanser til objektene funksjonen kalles med.
void getUserInput(double& theta, double& absVelocity);

// 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. Komponentene oppdateres gjennom referansene.
// med Vector i funksjonsnavnet tenker vi på vektor-begrepet i geometri
void getVelocityVector(double theta, double absVelocity,
double& velocityX, double& velocityY);

```

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:

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

Tilsvarende for `getVelocityY`:

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

Funksjonen `getVelocityVector()` kaller `getVelocityX()` og `getVelocityY()` og lagrer returverdiene fra disse funksjonene gjennom referansene `velocityX` og `velocityY`.

#### 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  $\text{posY} = 0$ .*

#### 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) dersom `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 skrive 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*).

### a) Opprett en ny fil, `utilities.cpp`, og tilhørende headerfil.

Filen skal inneholde en funksjon for å generere pseudotilfeldige tall.

#### Nyttig å vite: `rand` og `srand`

Det finnes i moderne C++ hovedsakelig to måter å generere tilfeldige tall på. Her skal vi bruke den «gamle» metoden, som kommer fra programmeringsspråket C, som er å bruke funksjonen `rand` fra biblioteket `cstdlib`. `rand` genererer pseudotilfeldige tall i intervallet 0 til `RAND_MAX`, der `RAND_MAX` er en konstant som er definert i biblioteket `cstdlib`. (Læreboka kapittel 24.7 presenterer `randint` som er en nyere og litt bedre måte å generere tilfeldige tall på, men vi skal ikke bruke det her).

Dersom man genererer to rekker med tall basert på samme frø, vil disse tallrekkenes være identiske. Siden `rand` ikke automatisk får et nytt frø hver gang programmet kjøres, er vi nødt til å gjøre dette manuelt for å unngå at `rand` alltid gir oss samme tallrekke. Å velge hvilket frø som brukes kalles å *initiere* (Eng. *seeding*) `rand`. Dette gjøres ved å bruke funksjonen `srand`, som også finnes i biblioteket `cstdlib`.

Det er vanlig å bruke returverdien fra `time(nullptr)` til å initiere, dvs. som input til `srand`. `time` finnes i biblioteket `ctime`. Denne verdien er av historiske årsaker antallet sekunder siden nyttår 1970 (og endrer seg derfor hvert sekund). Dersom man kjører `srand` med `time(nullptr)`, vil `rand` altså gi en unik tallrekke hver gang programmet kjøres forutsatt at man ikke starter programmet to ganger i løpet av samme sekund. Merk at man kun skal initiere én gang hver gang programmet kjøres. Her er et eksempel på bruk av `rand` og `srand`:

```
srand(static_cast<unsigned int>(time(nullptr)));
int tilfeldig_tall = rand();
// tilfeldig_tall inneholder nå et tilfeldig
// heltall mellom 0 og RAND_MAX
```

`srand` forventer en `unsigned int` som parameter og vi må derfor utføre en eksplisitt typekonvertering `static_cast` på den verdien `time` returnerer.

- b) Skriv en funksjon `randomWithLimits` som tar inn en øvre og nedre grense, og som bruker `rand` til å returnere et heltall i intervallet gitt av disse grensene (grensene skal være med i intervallet). 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 `srand` med `time`**. Test deretter programmet ditt igjen, som beskrevet i forrige deloppgave. *Hint: Bruk `srand` som beskrevet over. Husk at `srand` kun skal kjøres én gang, som betyr at den bør kjøres fra `main`-funksjonen.*
- d) Implementer funksjonen `playTargetPractice()`.

```
void playTargetPractice();
```

`playTargetPractice()` skal lese inn vinkel og startfart fra brukeren. Deretter skal funksjonen skyte med kanonen mot en tilfeldig plassert blink, mellom 100 og 1000 meter, og si hvor langt unna spilleren var fra å treffe målet. Spilleren skal få 10 forsøk på å treffe blinken og for hvert forsøk skal avstanden til målet, samt om skuddet var for langt eller for kort, skrives til skjerm. Det skal også oppgis hvor lang tid kulen har brukt på reisen, på en måte det er lett å tolke 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 10 forsøk. Hvis spilleren vinner skal hun gratuleres, og tilsvarende skal hun informeres om tap dersom det motsatte skjer.

#### e) Visualisering med FLTK — frivillig —

Dette er en frivillig oppgave som krever at du har en riktig installasjon av FLTK og grafikkbiblioteket som følger med læreboka. **Se Øving 0 Blackboard for nærmere detaljer.** Det betyr at denne oppgaven krever at du oppretter et grafikkprosjekt. Du kan kopiere alle filene så langt i øvingen over i det nye prosjektet, eller legge til filene fra prosjektet du har arbeidet med til nå som allerede eksisterende filer.

I den utdelte koden, `cannonball_viz.h` og `cannonball_viz.cpp`, 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 stegvis tegner banen. `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. Funksjonen er laget slik at man må trykke på Next-knappen oppe til høyre i vinduet for å vise hvert tidssteg, prøv gjerne med ulike antall tidssteg. Kallet på funksjonen skal gjøres en gang inne i den løkka der du gir tekstlig feedback til spilleren.

Du kan oppdage at konsollvinduet forsvinner etter at grafikkvinduet lukkes. Det kan du gjøre noe med ved å kalle funksjonen `keep_window_open()`. Se forelesning 1 og bokens side 53, nest siste avsnitt. Funksjonen er definert i headerfilen `std_lib_facilities.h`.