



Norges teknisk-naturvitenskapelige  
universitet  
Institutt for datateknikk og  
informasjonsvitenskap

TDT4102 Prosedyre  
og Objektorientert  
programmering  
Vår 2015

Øving 2

**Frist: 2015-01-30**

### Mål for denne øvinga:

- Lære å skrive koden din i flere filer
- Lære forskjellene mellom vanlige variabler og pekere
- Lære å bruke nyttige funksjoner fra standardbiblioteket som `sin` og `cos`
- Lære å bruke pseudotilfeldige tall vha `rand` og `srand`

### Generelle krav:

- Bruk de eksakte navn og spesifikasjoner som er gitt i oppgava
- Det er valgfritt om du vil bruke en IDE (Visual Studio, Xcode), men koden må være enkel å lese, compilere og kjøre
- Dersom noe er uklart eller trenger en bedre forklaring ta kontakt med en studass på sal.

### Anbefalt lesestoff:

- Kapittel 3 & 4, Absolute C++ (Walter Savitch)

## Å skrive kode i flere filer:

Når man programmerer er det viktig å være ryddig. Vanligvis vil koden til et C++-program være strukturert som dette:

- En «main»-fil
- En eller flere «.h»-filer (Headerfiler)
- En eller flere «.cpp»-filer (Implementasjonsfiler)

### Hoved- / «main»-fila

Dette er fila som kjøres når programmet startes. Utførelsen av programmet vil begynne i en funksjon som heter `main`. Dette funksjonsnavnet er standard for alle C++-programmer.

### Headerfiler

Disse filene inneholder informasjon som gjør det mulig å bruke den koden du skriver uten å vite hvordan den egentlig fungerer. Kan sees på som en enkel brukermanual for programmet.

Filene inneholder normalt funksjonsprototypene til alle funksjoner som skal være tilgjengelig/mulig å bruke for andre/utenforstående kode.

### Implementasjonsfiler

I disse filene skal resten av koden ligge. Dette er vanligvis her man skriver mesteparten av koden. I motsetning til headerfiler som kun sier noe om hvordan man kan bruke koden, vil disse filene inneholde implementasjonen.

### Filenes relasjon til hverandre:

Headerfiler skal inkluderes av både «main»-fila og implementasjonsfiler som bruker funksjoner beskrevet i headerfilen. Dette gjør at funksjonene som er beskrevet er tilgjengelige fra «main»-funksjonen og det sørger også for at koden i implementasjonsfilene blir ryddigere (vi kommer tilbake til dette). Konvensjonen er at hver .cpp-fil (kanskje med unntak av den som inneholder `main`-funksjonen) har en headerfil med samme navn, som beskriver hva som er implementert av den tilsvarende .cpp-filen.

Dvs., har man en fil som heter «a.cpp» har man også «a.h». For å finne ut hva som er implementert i «a.cpp» kan man se på «a.h».

## Oppgaveforklaring:

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

## Del 1: Enkel bevegelse:

### 1 Funksjonsprototyper (5%)

I denne oppgava skal vi lage et sett med funksjonsprototyper. Funksjonsprototypene vil bli beskrevet med tekst og det er opp til deg å tolke dette og skrive passende funksjonsprototyper. Alle funksjonsprototypene skal være i en egen «header»-fil. I denne øvinga skal den hete «`cannonball.h`». Det er vanlig at man grupperer funksjoner og funksjonalitet som hører sammen i *biblioteker* som vi kan bruke senere uten å vite hvordan de fungerer internt.

De påfølgende funksjonsprototypene danner grunnlaget for et *bibliotek* som kan regne ut banen til ei kanonkule. Oppgava er strengt spesifisert og det skal ikke være nødvendig å gjøre antagelser. Om du likevel føler at det er noe som er tvetydig eller dårlig spesifisert noter gjerne dette i koden din (kommentarer i C++ kan skrives som `/* */`) og ta det opp med din studentassistent på sal.

#### Eksempel-deloppgave:

For den første deloppgaven (1a) er løsningen allerede gitt. Man trenger kun å kopiere denne over til headerfilen, da du får bruk for den i senere oppgaver.

- a) **Lag en funksjonsprototype for en funksjon som returnerer akselerasjonen i Y-retning (oppover).**

Akselerasjonen i Y-retning er normalt et desimaltall. Denne funksjonen skal hete «`accLY`».

*Løsning:*

```
double accLY();
```

*Husk at ingen av de påfølgende deloppgavene forventer noe mer en bare funksjonsprototypen som i eksempelet over.*

- b) **Skriv en funksjonsprototype for en funksjon som regner ut farten i y-retning (oppover).**

Denne funksjonen tar inn to flyttall (double): startfart (`initVelocity`) og tid (`time`). Til slutt returnerer funksjonen farten som et flyttall. Funksjonen skal hete «`velY`».

- c) **I denne deloppgava skal vi lage et sett med funksjonsprototyper for funksjoner som regner ut posisjon i henholdsvis X- og Y-retning.**

Vi trenger en funksjon for hver retning, så dette gir 2 funksjonsprototyper:

Posisjon i X-retning

Posisjon i Y-retning

Begge disse tar inn to flyttall hver: startfart (`initVelocity`) og tid (`time`). Vi går ut i fra at startposisjonen er 0 (så denne trenger vi ikke å ta inn).

- d) **Skriv en funksjonsprototype for en funksjon som tar inn tid i sekunder og ikke returnerer noe.**

Denne skal hete «`printTime`».

- e) **Skriv en funksjonsprototype for en funksjon som tar inn startfarten i y-retning og returnere flytiden i sekunder.**

Denne skal hete «`flightTime`».

## 2 Implementer funksjoner (15%)

I denne oppgava skal vi implementere funksjonene fra forrige oppgave. Alle funksjonsimplementasjoner skal ligge i en implementasjonsfil tilhørende «header»-fila. I dette tilfellet heter den «cannonball.cpp» siden vi skal implementere funksjonen som er lagt i «cannonball.h».

Formler som trengs vil bli presentert. Din oppgave er å skrive en funksjon som bruker formelen og passer til funksjonsprototypen som vi laget tidligere. Det kan lønne seg å kopiere funksjonsprototypene fra «header»-fila som en start da funksjonene må passe funksjonsprototypene eksakt for at det skal fungere. Husk også å inkludere «header»-fila i implementasjonsfila, dette kan gjøres ved:

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

Det er viktig at begge filene, «cannonball.h» og «cannonball.cpp» ligger i samme mappe for at dette skal fungere.

- a) I denne oppgava skal du implementere funksjonen fra oppgave 1a. Denne funksjonen returnerer akselerasjonen i y-retning (oppover).

Til vanlig er akselerasjonen i y-retning  $-9.81\text{m/s}$  (gjenstander trekkes mot bakken).

- b) Denne oppgava skal implementere funksjonen fra oppgave 1b (fart i y-retning).

Funksjonen gjør følgende utregning basert på de verdiene den får inn:

$$fartY = startFartY + akselY * tid \quad (1)$$

- c) I denne oppgava skal vi lage / implementere funksjonene som regner ut posisjonen i X- og Y-retning (oppgave 1d).

Formelen for dette er:

$$posisjon = startPosisjon + startFart \cdot tid + \frac{akselerasjon \cdot tid^2}{2} \quad (2)$$

- d) Implementer funksjonen «printTime» (oppgave 1e).

Vi ønsker å dele opp sekundene i timer, minutter og sekunder (sekunder kan evt være et desimaltall) for så å skrive dette til skjerm.

*Hint: se hvor mange timer det er først.*

- e) Implementer funksjonen «flightTime» (oppgave 1f).

«FlightTime» skal finne ut hvor lenge noe kommer til å fly (i dette tilfelle kanonkula) gitt en fart. Det er kun farten i Y-retning som har noe å si siden vi går ut ifra en perfekt flat bakke og ingen luftmotstand etc. Forestill deg følgende: ei kule som kastes rett opp mister etter hvert farten på grunn av tyngdekraften. Når den er på det høyeste vil farten være 0, deretter vil den begynne å falle. Når kula igjen er ved bakken vil farten være like stor som den var ved starten av kastet, men motsatt rettet (altså nedover mot bakken).

### 3 Verifiser at funksjonene fungerer (10%)

Det er veldig lurt å teste koden litt etter litt. Ofte kan det være utfordrende å finne feil dersom man kjører større programmer sammen da man ikke vet hvor feilen oppstår. Når man tester C++ kode ønsker vi å finne ut følgende:

- Kompilerer programmet? Dette luker ut de fleste syntaksfeil og enkelte grove logiske feil.
- Gjør programmet det du forventer at det skal gjøre? For å teste dette må vi vite noe om hva programmet skal gjøre dersom det fungerer.

#### a) Forsikre deg om at programmet kompilerer.

Dersom programmet ikke kompilerer sjekk følgende:

- Programmet har følgende filer: «main.cpp», «cannonball.h» og «cannonball.cpp»
- Filene er satt opp som følger: «main.cpp» skal inkludere «cannonball.h». «cannonball.cpp» skal inkludere «cannonball.h»
- I filen «main.cpp» finnes det én main funksjon («int main()»)
- .cpp-filer skal ALDRI inkluderes, kun .h-filer.

Dersom det fortsatt ikke fungerer når du prøver å kompilere må du se på feilmeldingene. Dersom du ikke forstår feilmeldingen kan Google og studentassistenten din være til hjelp.

#### b) Test hver funksjon fra main-funksjonen.

Hver funksjon bør testes, dette kan gjøres enkelt ved å bruke et sett av eksempeldata der man vet svaret på forhånd. For eksempelverdier se tabellen under:

	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	250.0
posY	0.0	31.84	2.375

I dette tilfellet kommer de fleste av testene til å involvere desimaltall. Desimaltall fungerer noe anderledes i en datamaskin enn det som er naturlig, dette kan du lese mer om på Wikipedia:

[http://en.wikipedia.org/wiki/Floating\\_point#Accuracy\\_problems](http://en.wikipedia.org/wiki/Floating_point#Accuracy_problems)

I denne oppgava holder det å si at det å sammenligne (dnumber1 == dnumber2) er en dårlig idé. Det er to grunner til dette:

1. Det finnes tilfeller der to tilsynelatende like tall (om du skriver de til skjerm så ser de like ut) ikke er representert på samme måte i datamaskinen og dermed vil bli sagt å være ulike! Dette er på grunn av avrunding.
2. På grunn av forskjeller i datamaskinens prosessor er det ikke garantert at en lik sammenligning vil returnere rett selv om de to tallene er like (fordi datamaskinen internt kan velge å lagre tallene på forskjellige måter).

Den beste måten å sjekke om svaret stemmer på vil være å teste om avviket fra det korrekte svaret et veldig lite, f.eks:

```

double forventetResultat = /* det forventede resultatet settes her */
double resultat = /* resultatet fra din funksjon settes her */
double avvik = pow(forventetResultat - resultat, 2.0);
if (avvik < feilmargin){
    /* Fantastisk, det var rett! */
}
else{
    /* Resultatet var galt, det falt utenfor feilmarginen */
    cout << "Funksjon X hadde et avvik paa " << avvik << endl;
}

```

## Del 2: Gjenbruk av funksjoner

### 4 Implementer funksjoner (15%)

a) I denne oppgava skal vi implementere følgende funksjoner:

```

// Ber brukeren om to tall, en vinkel, og en fart.
// Disse verdiene skal plasseres i minnet som pekerene
// theta og absVelocity peker paa.
void getUserInput(double *theta, double *absVelocity);

// Gir 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, disse komponentene plasseres i minnet
void getVelocityVector(double theta, double absVelocity,
    double *velocityX, double *velocityY);

```

Disse funksjonene brukes til å lese inn en vinkel og en fart fra brukeren. For å kunne returnere mer enn en verdi kan vi benytte oss av pekere. Pekere er som navnet tilsier en peker til en plass i minnet til datamaskinen. Fordelen med å jobbe med pekere er at funksjonen som får pekeren har mulighet til å gå inn i minnet og endre verdien slik at endringen også blir gjeldende utenfor funksjonen. Syntaks for pekere som funksjonsargument kan sees i «getUserInput». «double \*» betyr her at denne tar inn en peker som peker til en plass i minnet der det finnes en «double»-variabel.

Når vi bruker pekere må vi skille mellom å endre minneadressen som den peker til og innholdet som blir pekt på, for å endre innholdet som pekes på gjør man følgende:

```

void foo(double *b){
    *b = 2.0;
}

```

For å endre minneadressen som blir pekt på gjør man dette (eksempelet nedenfor setter pekeren til NULL): I dette eksempelet vil adressen bli kopiert til en ny lokal variabel «b».

```
void foo(double *b){
    b = NULL;
}
```

Det gjør at endringer på «b», det vil si **adressen** «b» peker til, vil ikke få noen effekt utenfor funksjonen! Funksjonen over gjør ingenting!

Ser man bort fra at man må skrive \*b istedenfor b, kan pekere brukes på samme måte som vanlige variabler. Når vi skal kjøre funksjonen som tar inn en peker kan vi gjøre følgende:

```
double a = 0.0;
foo(&a);
```

Dette gjør at vi sender inn adressen der «a» er lagret fremfor verdien «a» har.

Funksjonen «getVelocityX» skal dekomponere absoluttfarten i X-retning, dette kan gjøres som følger:

$$fartX = absFart \cdot \cos(vinkel) \quad (3)$$

Tilsvarende gjøres for «getVelocityY»:

$$fartY = absFart \cdot \sin(vinkel) \quad (4)$$

Funksjonen «getVelocityVector» kjører begge funksjonene («getVelocityX» og «getVelocityY») og gir tilbake resultatene gjennom pekerene «velocityX» og «velocityY».

**b) Implementer funksjonen «getDistanceTraveled»**

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

Funksjonen «getDistanceTraveled» skal returnere avstanden som kanonkula reiste før den traff bakken, med andre ord hva «Posisjon X» er når «Posisjon Y» er 0. Den enkleste måten å løse dette på er ved å bruke funksjonen «flightTime».

**c) Implementer funksjonen «targetPractice».** Funksjonen skal ta inn en avstand «distanceToTarget» og returnere avviket i meter mellom der kulen distanceToTarget og der kulen lander dersom «velocityX» og «velocityY» er henholdsvis startfart i X- og Y-retning.

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

## 5 Verifiser at funksjonene fungerer (10%)

**a) Verifiser at programmet kompilerer**

*Tips: Har du lagt til funksjonsprototypene i «header»-fila?*

**b) Lag en kodesnutt i «main()» som tester funksjonene**

## 6 Tilfeldige tall (10%)

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

Dettee biblioteket skal inneholde en funksjon som gir oss et pseudotilfeldig tall.

- b) Skriv en funksjon `randomWithLimits` som tar inn en øvre og nedre grense, og returnerer et tilfeldig heltall mellom disse. (inklusive grenser) Forsikre deg om at denne funksjonen fungerer slik du har tenkt ved å teste den. Bruk gjerne en løkke for å kjøre funksjonen flere ganger, og kjør programmet flere ganger.

*Hint: Inkluder `cstdlib` for å bruke `rand`-funksjonen.*

- c) Funksjonen over gir samme resultat ved hver kjøring, gjør slik at du får forskjellig resultat ved hver kjøring

Dette kan fikses ved å legge til følgende helt først i `main`:

```
srand(time(0));
```

Tilfeldige tall i C++ er bare pseudotilfeldige, og regnes ut med en formel. Man kan tenke seg at man plukker kort med tall på fra en kortstokk som er stokker på akkurat samme måte hver gang programmet starter. For å stokke om på sekvensen av tall kan man bruke `srand()`-funksjonen, som også finnes i `cstdlib`. `srand()` står for «Seed RANdom (number generator)». Det er rimelig vanlig å bruke resultatet fra `time(0)` (som av historiske årsaker returnerer antallet sekunder siden nyttår 1970) for å få en helt ny sekvens med pseudotilfeldige tall hver gang man kjører programmet (forutsatt at man ikke starter programmet to ganger samme sekund).

*Hint: Inkluder `ctime` for å bruke `time`-funksjonen.*

## 7 Større program (35%)

- a) Legg til funksjonen «`playTargetPractice`» i «`cannonball.cpp`» (og oppdater den tilsvarende headerfilen) og legg til kode i «`main()`» for å kjøre denne funksjonen.

```
void playTargetPractice();
```

Vi ønsker i denne oppgava å sette sammen alle funksjonene til et større program. Dette programmet skal ta inn en vinkel og fart fra brukeren. Deretter skal programmet skyte med kanonene mot et tilfeldig plassert (mellom 100m og 1000m) mål og si hvor langt unna brukeren var fra å treffe målet. Spilleren skal få 10 forsøk på å treffe målet og for hvert forsøk skal marginen samt om skuddet var for langt eller for kort skrives til skjerm.

Dersom kanonkulen lander mindre enn fem meter unna målet regnes det som om man treffer, og at spilleren har vunnet. Man taper dersom man bruker opp alle ti forsøk.