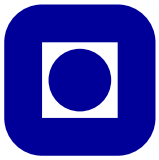

Norges teknisk–naturvitenskapelige universitet
Institutt for datateknikk og informasjonsvitenskap

TDT4102 Prosedyre- og objektorientert programmering Vår 2016

Øving 3

Frist: 2016-02-05

Mål for denne øvingen:

- 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 om to metoder å generere tilfeldige tall på, og når de brukes

Generelle krav:

- Bruk de eksakte navn og spesifikasjoner som er gitt i oppgaven
- Det er valgfritt om du vil bruke en IDE (Visual Studio, Xcode), men koden må være enkel å lese, kompilere 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 «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 implementerer man 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 inneholder funksjonsprototyper for alle funksjoner som skal gjøres tilgjengelige når headerfilen inkluderes fra en implementasjonsfil. Man kan også bruke headerfilene som en slags enkel «brukermanual» for programmet, da man enkelt kan bruke dem til å se hvilke funksjoner som er tilgjengelige uten å måtte grave seg ned i implementasjonsfilene.

Filenes relasjon til hverandre:

Headerfiler skal inkluderes av alle implementasjonsfiler, også «hovedfilen», som bruker funksjoner beskrevet i den aktuelle headerfilen. Når man inkluderer en headerfil i en implementasjonsfil blir funksjonene som er beskrevet i headerfilen tilgjengelig for bruk i implementasjonsfilen.

Konvensjonen er at hver `.cpp`-fil – kanskje 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 altså man også `minefunksjoner.h`. For å finne ut hva som er implementert i `minefunksjoner.cpp` kan man se på `minefunksjoner.h`.

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.

Del 1: Enkel bevegelse

1 Funksjonsprototyper (5%)

I denne oppgaven skal vi lage et sett med funksjonsprototyper. Funksjonsprototypene vil bli beskrevet med tekst og det er opp til deg å tolke denne og skrive passende funksjonsprototyper. Alle funksjonsprototypene skal være i en egen header-fil., som vi i denne øvingen skal kalle `cannonball.h`.

Funksjonsprototypene du skal lage i denne oppgaven danner grunnlaget for et *bibliotek* som kan regne ut banen til en kanonkule. Oppgaven 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 legg gjerne til en kommentar om dette i koden din (kommentarer i C++ kan skrives som `/* */` eller med `//` foran hver linje) og ta det opp med din studentassistent på sal.

Eksempel-deloppgave:

For den første deloppgaven, 1 a), er løsningen allerede gitt. Kopier likevel denne over til header-filen, 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 skal være et flyttall (`double`). 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 (`initVelocityY`) og tid i sekunder (`time`). Til slutt returnerer funksjonen farten som et flyttall. Funksjonen skal hete `velY`.

c) I denne deloppgaven 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 (`double`) hver: startfart (`initVelocity`) og tid (`time`). Vi antar at startposisjonen er 0, så denne skal ikke tas inn. Kall funksjonene `posX` og `posY`.

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 returnerer flytiden i sekunder.
Denne skal hete `flightTime`.

2 Implementer funksjoner (15%)

I denne oppgaven skal vi implementere funksjonene fra forrige oppgave. Alle funksjonsimplementasjoner skal ligge i en implementasjonsfil tilhørende header-filen. I dette tilfellet heter den `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 funksjonsprototypene vi lagde tidligere. Det kan lønne seg å kopiere funksjonsprototypene fra header-filen som en start da funksjonene må passe funksjonsprototypene eksakt for at koden skal fungere. Husk også å inkludere header-filen i implementasjonsfilen. Dette gjøres ved å skrive følgende:

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

Det er viktig at `cannonball.h` og `cannonball.cpp` ligger i samme mappe for at dette skal fungere.

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

Til vanlig er den vertikale akselerasjonen med positiv retning oppover -9.81m/s (gjenstander trekkes mot bakken).

b) Denne oppgaven skal implementere funksjonen beskrevet i Oppgave 1 b) (fart i Y-retning).

Funksjonen skal gjøre følgende utregning basert på de verdiene den får inn:

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

c) I denne oppgaven skal vi lage og implementere funksjonene fra Oppgave 1 c), som regner ut posisjonen i X- og Y-retning.

Formelen for dette er:

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

d) Implementer funksjonen fra Oppgave 1 d), `printTime`.

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

Hint: Se hvor mange timer det er først, og så videre.

e) Implementer funksjonen fra Oppgave 1 f), `flightTime`.

`flightTime` skal finne ut hvor lenge det aktuelle objektet, i dette tilfelle en kanonkule, kommer til å fly gitt en startfart. I denne oppgaven antar vi en perfekt flat bakke og at det kun er tyngdekraften som påvirker kulens fart, slik at det kun vil være startfarten i Y-retning som vil påvirke flytiden.

Forestill deg følgende: en kule som kastes rett opp mister etter hvert farten på grunn av tyngdekraften. Når den er på det høyeste vil farten i Y-retning være 0, og deretter vil den begynne å falle. Når kulen 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 skriver mye kode før man tester den, da det vil bli veldig mange muligheter for hvor feilen kan ha oppstått. Når vi 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 hva det er meningen at programmet skal gjøre hvis 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 nærmere på feilmeldingene. Forstår feilmeldingen kan du prøve å søke etter den med Google, spørre studentassistenten din eller spørre på emnets diskusjonsforum på It's learning.

b) Test hver funksjon fra main-funksjonen.

Hver funksjon bør testes. Dette kan enkelt gjøres ved å bruke et sett av eksempeldata der man vet svaret på forhånd, og så sjekke om svaret man får fra funksjonen stemmer med eksempeldataen. Noen eksempelverdier finnes i følgende tabell:

	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. Vi representerer desimaltall ved hjelp av flyttall, men disse oppfører seg imidlertid ikke alltid på samme måte som vi forventer at desimaltall skal. Dette kan du blant annet lese mer om i Wikipedia-artikkelen om flyttall.

I denne oppgaven holder det å si at det å sammenligne flyttall direkte (`flyttall1 == flyttall2`) er en dårlig idé. Det skjer ofte at to tall som «skal» være like i henhold til vanlig aritmetikk med desimaltall representeres på ulik måte som flyttall, og da vil en sammenligning gi feil svar. Den beste måten å sjekke om to flyttall er like på er å teste om avviket mellom dem er veldig lite. Et eksempel:

```
double forventetResultat = /* det forventede resultatet settes her */
double resultat = /* resultatet fra din funksjon settes her */
double avvik = std::pow(forventetResultat - resultat, 2.0);

if (avvik < feilmargin) {
    /* Fantastisk, det var rett! */
} else {
    /* Resultatet var galt, det falt utenfor feilmarginen */
    std::cout << "Funksjon X hadde et avvik paa " << avvik << std::endl;
}
```

Del 2: Gjenbruk av funksjoner

4 Implementer funksjoner (20%)

a) I denne oppgaven skal vi implementere følgende funksjoner:

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

// 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. Disse komponentene lagres 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 og omgjøre de innleste verdiene til hastighet i X- og Y-retning. Den siste funksjonen, `getVelocityVector`, kombinerer så de tre foregående funksjonene i én funksjon.

I C++ kan vi i utgangspunktet kun returnere én verdi fra en gitt funksjon. For å omgå denne begrensningen kan vi benytte oss av pekere. En peker er, som navnet tilsier, en henvisning til en plassering eller *adresse* i datamaskinens minne. Hvis vi sender inn en peker som argument til en funksjon, kan funksjonen som mottar pekeren gå inn i minnet og *endre* på verdien som ligger lagret der. Den nye verdien pekeren henviser til vil da også være gjeldende utenfor funksjonen, etter at funksjonen er ferdig.

Et eksempel på en funksjonsprototype til en funksjon som tar inn pekere som argumenter kan sees i prototypen til `getUserInput` over. `double *` betyr her at funksjonen tar inn en peker som henviser til en plassering i minnet der det finnes en `double`-variabel. For å hente ut en verdi fra en gitt peker, benytter vi asterisk-operatoren (`*`) foran navnet på pekeren. I følgende eksempel henter vi ut verdien av en peker og skriver denne ut til skjerm: Når

```
void foo(double *b){
    std::cout << *b << std::endl;
}
```

vi bruker pekere må vi skille mellom å endre *minneaddressen* som pekeren henviser til, og *innholdet* som blir pekt på. For å endre innholdet som pekes på gjør man følgende: For

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

å endre minneaddressen som blir pekt på gjør man følgende: I dette eksempelet endrer vi

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

pekeren `b` slik at den peker på samme adresse i minnet som `a`. Dette vil imidlertid ikke ha noen effekt *utenfor* funksjonen `foo`, siden vi ikke har endret på hva som finnes på adressen som `b` pekte til – kun pekeren selv, og denne påvirker ikke eventuelle pekere til samme minnelokasjon som finnes utenfor funksjonen.

Ønsker vi å sende en variabel inn i en funksjon som tar inn en peker, kan vi gjøre følgende: Vi sender her inn en *referanse* til adressen der `a` er lagret. Denne referansen vil fungere

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

som en peker.

I funksjonen `getVelocityX` skal vi beregne farten i X-retning. Dette kan gjøres som følger:

$$\text{FartX} = \text{AbsoluttFart} \cdot \cos(\text{Vinkel}) \tag{3}$$

Tilsvarende gjøres for `getVelocityY`:

$$\text{FartY} = \text{AbsoluttFart} \cdot \sin(\text{Vinkel}) \tag{4}$$

Funksjonen `getVelocityVector` kjører begge funksjonene (`getVelocityX` og `getVelocityY`) og gir tilbake returverdiene fra disse funksjonene gjennom pekerene `velocityX` og `velocityY`.

Hint: Funksjoner for sinus og cosinus finnes i biblioteket `cmath`.

b) Implementer funksjonen `getDistanceTraveled`:

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

Funksjonen `getDistanceTraveled` skal returnere den horisontale avstanden kanonkulen reiste før den traff bakken – med andre ord verdien til posisjonen i X-retning når posisjonen i Y-retning (høyde over bakken) er 0. Den enkleste måten å løse dette på er ved å bruke funksjonen `flightTime` som du lagde tidligere.

c) Implementer funksjonen `targetPractice`. Funksjonen 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) Lag en kodesnutt i `main()` som tester funksjonene du har implementert. Verifiser at programmet ditt kompilerer, og at det fungerer slik du forventer.

Tips: Har du lagt til funksjonsprototypene i header-filen?

5 Tilfeldige tall (15%)

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 visst *intervall*, som vil si at alle de mulige tallene i intervallet skal være like sannsynlige. At vi genererer *pseudotilfeldige* tall betyr at tallene regnes ut med en deterministisk algoritme basert på en «startverdi» som kalles en *seed*.

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

Det finnes i moderne C++ hovedsakelig to måter å generere tilfeldige tall på. Den «gamle» metoden, som kommer fra programmeringsspråket C, er å bruke funksjonen `std::rand` fra biblioteket `cstdlib`. `std::rand` genererer pseudotilfeldige tall i intervallet 0 til RAND_MAX, der RAND_MAX er en konstant som er definert i biblioteket `cstdlib`.

b) Skriv en funksjon `randomWithLimits` som tar inn en øvre og nedre grense, og som bruker `std::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 gjerne en løkke for å kjøre funksjonen flere ganger. Kjør programmet flere ganger og se hva som skjer.

Dersom man genererer to rekker med tall basert på samme seed, vil disse tallrekkene være identiske. Siden `std::rand` ikke automatisk seedes med et nytt tall hver gang programmet kjøres, er vi nødt til å gjøre dette manuelt for å unngå at `std::rand` alltid gir oss samme tallrekke. Å velge hvilken seed som brukes kalles å «seede» `std::rand`. Dette gjøres ved å bruke funksjonen `std::srand`, som også finnes i biblioteket `cstdlib`.

Det er rimelig vanlig å bruke returverdien fra `std::time(NULL)` til å seede, dvs. som input til `std::srand`. `std::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 `std::srand` med `std::time(NULL)`, vil `std::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 seede én gang hver gang programmet kjøres. Her er et eksempel på bruk av `std::rand` og `std::srand`:

```
std::srand(std::time(NULL));
int tilfeldig_tall = std::rand();
// tilfeldig_tall inneholder nå et tilfeldig
// heltall mellom 0 og RAND_MAX
```

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 `std::srand` med `std::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 må kjøres fra `main`-funksjonen.*

Dessverre er det enkelte feller man kan gå i ved bruk av `std::rand`.

- `std::rand` er laget for å være rask, men genererer ikke nødvendigvis tilfeldige tall av «høy kvalitet» (for eksempel er ikke tallene som blir generert nødvendigvis uavhengige av hverandre). Dette gjør at `std::rand` ikke kan brukes til kryptografi og en del vitenskapelige formål.
- Vi ønsker vanligvis å generere et tall fra et visst intervall. For å generere et tilfeldig tall i intervallet $[0, \ldots, N-1]$ skriver vi ofte `rand() % N`. Dessverre vil distribusjonen nå *ikke* være uniform lenger: små tall være litt mer sannsynlige enn store, som kan føre til gale utregninger eller uventet oppførsel.

For å bøte på dette finnes det i moderne C++ en mer «idiotsikker» måte å generere tilfeldige tall på, i biblioteket `random`. Her er et eksempel på generering av et tilfeldig tall fra et intervall med grenser definert av variablene `MIN` og `MAX`:

```
1 std::random_device device;
2 std::mt19937 generator(device());
3 std::uniform_int_distribution<int> uniform(MIN, MAX);
4
5 int random_integer = uniform(generator);
```

I linje 1 til 3 setter vi opp en tilfeldig tall-generator basert på algoritmen MT19937, som er en vanlig algoritme for generering av pseudotilfeldige tall. Merk at vi ikke trenger å spesifisere en seed-verdi, da generatoren blir automatisk seedet. I linje 5 lager vi så et tilfeldig tall og lagrer det i variabelen `random_integer`. Uttrykket `uniform(generator)` kan gjentas for å generere flere tilfeldige tall.

d) Skriv en funksjon `modernRandomWithLimits` som tar inn en øvre og nedre grense, og bruker den moderne metoden beskrevet over (med bruk av `std::mt19937` osv.) for å returnere et tilfeldig heltall mellom de gitte grensene (inklusive grensene). Forsikre deg igjen om at funksjonen fungerer slik den skal ved å kjøre den i løkke og kjøre programmet flere ganger.

Hvilken metode bør vi så bruke til vanlig? Til vitenskapelige formål og til oppgaver innen sikkerhet og kryptografi må den moderne metoden brukes, da å bruke `std::rand` kan føre til sikkerhetshull eller gale utregninger. Ofte er kvaliteten til det tilfeldige tallet imidlertid ikke så viktig, for eksempel hvis man vil velge tilfeldig blant noen få mulige hendelser i et spill. Da kan det være like greit å bruke `std::rand`, så lenge man er klar over fellene det er mulig å gå i. Dessuten kan `std::rand` være noe raskere, og biblioteket `random` er ikke nødvendigvis tilgjengelig på alle platformer.

6 Større program (35%)

Vi ønsker i denne oppgaven å sette sammen alle funksjonene til en større funksjon som lar oss spille et «spill» der vi prøver å treffe en blink.

a) Implementer en funksjon `playTargetPractice` i `cannonball.cpp` og legg til kode i `main()` for å kjøre denne funksjonen. Husk å oppdatere `cannonball.h` med funksjonsprototypen.

```
void playTargetPractice();
```

`playTargetPractice` skal ta inn en vinkel og en startfart fra brukeren. Deretter skal funksjonen skyte med kanonene mot en tilfeldig plassert blink (mellom 100 og 1000 meters avstand) 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.

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 de ti forsøkene. Hvis spilleren vinner skal det skrives ut en gratulasjonsmelding til skjerm, og tilsvarende skal spilleren informeres om at han har tapt dersom han bruker opp alle forsøkene.

For å gjøre denne oppgaven må du generere et tilfeldig tall som representerer avstanden til målet. Her kan du bruke hvilken som helst av metodene du lærte i Oppgave 5.