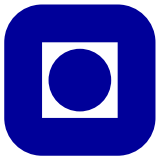

Norges teknisk–naturvitenskapelige universitet
Institutt for datateknikk og informasjonsvitenskap

TDT4102 Prosedyre- og objektorientert programmering Vår 2016

Øving 1 (Matlab)

Frist: 2016-01-22

Mål for denne øvingen:

- Overføre programmeringskonsepter vi kjenner til fra Matlab til C++

Generelle krav:

- Bruk de eksakte navn og spesifikasjoner 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.

Denne øvingen er tenkt som ekstra hjelp til de av dere som har noe programmeringserfaring fra før, men fra et annet språk enn C++. Gjennom enkle eksempler ønsker vi å vise likhetstrekkene mellom det du kan fra før og C++.

I denne øvingen skal vi demonstrere likheter og ulikheter mellom C++ og Matlab. Oppgavene vil i stor grad gå ut på å oversette Matlab-kode til C++, og kan kanskje være til hjelp som en slags oppfriskning av ITGK-kunnskapene. Samtidig vil de gi en mulighet til å overføre kunnskapen man har fra Matlab over til C++. Det vil også være viktig å lære seg å legge merke til de fallgruvene og forskjellene som finnes mellom de to språkene.

Lynintroduksjon til forskjellene mellom Matlab og C++

Generelt rammeverk

Siden vi ikke har gått så mye i dybden på C++ ennå, er det verdt å nevne enkelte ting som vi som regel trenger for å ha et gyldig program i C++. Kildekoden til et (nesten) minimalt C++-program, som ikke gjør noe som helst, ser slik ut:

```
1 #include <iostream>
2
3 int main() {
4     // Kode her
5     return 0;
6 }
```

En kort gjennomgang av hva som foregår her:

- Linje 1: Vi sier at vi trenger `iostream`-biblioteket, som blant annet inneholder funksjonalitet for utskrift til skjerm og input fra bruker. Et annet interessant bibliotek er f.eks. `cmath`, som inneholder sinus/cosinus/tangens, kvadratrot/potensfunksjoner og andre mattefunksjoner.
- Linje 3: Vi deklarerer `main`-funksjonen. Når du har kompilert og kjører programmet ditt vil det alltid starte med å utføre `main`-funksjonen. Legg merke til krøllparentesene, som markerer «kroppen» til funksjonen. Vi ser videre at `main`-funksjonen har *returtypen* `int`, som betyr at den kommer til å returnere et heltall. I C++ skal `main`-funksjonen alltid ha returtypen `int`.
- Linje 4: Denne linjen inneholder en kommentar. Disse starter med //, og sier at resten av linjen skal ignoreres. Som kommentaren sier er det her, mellom krøllparentesene, at koden til funksjonen skrives.
- Linje 5: Vi returnerer verdien 0 fra `main`. Dette er konvensjon for å si at programmet fullførte uten feil. I motsatt fall kan vi returnere verdien 1 for å indikere at programmet feilet. Legg merke til at verdien vi returnerer har samme type som vi deklarerte at main() funksjonen skulle returnere.

Du legger kanskje merke til at linje 4 og 5 er *indentert*. I C++, som i de fleste programmeringsspråk, er det vanlig å indentere innholdet i *blokker*, det vil si alt som er mellom krøllparentesene etter en `if`-setning, funksjonsdeklarasjon, `for`-loop, og så videre. Dette gjøres for at kildekoden skal være oversiktlig, og slik at man lettere forstår hvilken kode som henger sammen. Hvor mye du vil indentere kan du velge selv – det er vanlig å bruke enten 4 mellomrom eller en tabulator (tegnet som blir lagt inn når du trykker `tab`-knappen i en vanlig teksteditor).

Utskrift til skjerm

I Matlab skriver man ut tekst til skjerm enten med `fprintf("A:%d B:%d C:%d", a,b,c)` eller med `disp(a, b, c)`. I C++ sender vi i stedet tekststrenger og andre objekter vi ønsker å skrive til skjerm til objektet `std::cout` (standard utstrøm) ved å bruke <<-operatoren. Det er tillatt å nøste denne og dermed skrive ut flere objekter om gangen. Siste linje i figuren under gir et eksempel på det.

`std::endl` er et forhåndsdefinert objekt som representerer linjeskift. I Matlab bruker man `\n` i `fprintf` for å eksplisitt skrive ut et linjeskift. Dette er også mulig å gjøre i C++ – `\n` inni en tekststreng som sendes til `std::cout` vil gi et linjeskift, på lik linje som i Matlab.

I Matlab er `disp` en innebygd funksjon som man kan bruke uten å importere noen biblioteker, mens for å bruke `std::endl` er man nødt til å inkludere `iostream`-biblioteket. Det er også verdt å si noe om navnet `std::endl`. Dette består av to deler, `std` og `endl`, adskilt av to kolon. Her er `endl` selve *funksjonsnavnet*, mens `std` er navnerommet (på engelsk *namespace*) som objektet `cout` befinner seg i. Et navnerom er en slags kategori av funksjoner som kan gå på tvers av ulike biblioteker. For at ikke navnene på funksjonene i standardbiblioteket skal kollidere med navnene på dine variabler og funksjoner, er alle funksjoner, variabler og objekter i standardbiblioteket plassert i navnerommet `std`.

Læreboken og foreleser vil av og til bruke en setningen `using namespace std` først i programmet for å slippe å skrive `std::` hver gang funksjoner fra standardbiblioteket brukes. Dette gjør programmene litt kortere, «penere» og lettere å lese. Dersom man bruker et annet bibliotek enn standardbiblioteket mye kan det også være nyttig å bruke `using namespace` for det biblioteket. Å benytte seg av `using namespace` er en grei praksis for små programmer, og de fleste vi skal jobbe med i dette faget. I større programsystemer unngås imidlertid ofte `using namespace` grunnet muligheten for navnekollisjoner, og mange programvareprosjekter og -bedrifter tillater ikke bruk av `using namespace`, deriblant Google.

```
disp(2+3)
disp('Hello world!')
fprintf('2+5=%d\n',2+5)
```

(a) Matlab

```
std::cout << 2+3 << std::endl;
std::cout << "Hello world!" << std::endl;
std::cout << "2+5=" << (2+5) << std::endl;
```

(b) C++

Figur 1: Utskrift til skjerm

Variabler

I Matlab kan man introdusere variabler bare ved å tilordne et variabelnavn en verdi. I C++ må vi første gang en variabel brukes deklarere den ved å skrive datatypen foran variabelnavnet. Det er vanlig å tilordne verdier til variabler samtidig som vi deklarerer dem, men det er ikke påbudt.

```
a = 1;
b = 2;
c = a + b;
d = fix(c / b);
disp(d);
d = d * b;
disp(d);

e = c;
f = e / 2;
disp(f);

g = c / b;
disp(g);
```

(a) Matlab

```
int a = 1;
int b = 2;
int c = a + b;
int d = c / b;
std::cout << d << std::endl;
d = d * b;
std::cout << d << std::endl;

double e = c;
double f = e / 2.0;
std::cout << f << std::endl;

double g = c / static_cast<double>(b);
std::cout << g << std::endl;
```

(b) C++

Figur 2: Variabeldeklarasjon og bruk

Vi har her sett på to forskjellige datatyper, `int` og `double`. `int` er en type som kun kan representere heltall. En variabel av typen `int` tar på de fleste platformer opp 32 bits i minnet, og kan dermed representere heltall mellom -2^{31} og $2^{31} - 1$. `double` er en type som inneholder *flyttall*, som brukes til å representere reelle tall. Ulikt en variabel av typen `int` er en variabel av typen `double` vanligvis 64 bits stor, og har en presisjon på mellom 15 og 17 sifre avhengig av størrelsen på tallet. En ting som er verdt å merke seg er at flyttall ikke kan representere alle reelle tall (da det er et uendelig antall av dem i alle intervaller), og uvettig bruk av flyttall kan dermed lett føre til «avrundingsfeil».

På linje 4 er det verdt å merke seg forskjellen mellom divisjonsoperatorene i Matlab og C++. I C++ gir den vanlige divisjonsoperatoren `/` *heltallsdivisjon* når den blir brukt til å dele to heltall. Dette betyr at svaret av divisjonen blir rundet ned til nærmeste heltall, og tilsvarer å bruke funksjonen `fix` på resultatet av divisjonen i Matlab. Dersom man ønsker å få et desimaltall (flyttall) som svar fra en divisjon mellom to heltall, er man nødt til å *caste* det ene tallet til en flyttallstype, f.eks. `double`, og så utføre divisjonen. Å caste en variabel betyr å gjøre den om til en annen type. Caster man for eksempel heltallet 4 til `double`, vil man få flyttallet 4.0. Caster man et flyttall, for eksempel 4.5 til `int`, vil man få heltallet 4. Et eksempel på casting er vist til slutt i figuren over. Det finnes ulike former for casting, men foreløpig trenger du bare å bry deg om `static_cast`.

Input fra bruker

Matlab kan ta inn data fra brukeren ved hjelp av `input()`. I C++ bruker man i stedet objektet `std::cin`, som fungerer på en lignende måte som utskrift til skjerm med `std::cout`.

```
i = input('Skriv inn et tall: ');
j = input('Skriv inn et tall: ');
fprintf('Summen av de to tallene: %d\n', i + j);
```

(a) Matlab

```
double i = 0.0;
double j = 0.0;
std::cout << "Skriv inn et tall:" << std::endl;
std::cin >> i;
std::cout << "Skriv inn et tall:" << std::endl;
std::cin >> j;
std::cout << "Summen av de to tallene: " << (i+j) << std::endl;
```

(b) C++

Figur 3: Input fra bruker

Matlab lar oss skrive ut en forespørsel til bruker om hva som skal skrives inn, mens vi i C++ vil måtte gjøre dette i to steg, først en utskrift, og så en forespørsel om input. Ved bruk av >>-operatoren tar C++ seg av å lese inn og tolke verdiens type. Dette fungerer slik at verdien som leses inn vil bli tolket som å ha samme type som variabelen den skal lagres i, altså `double` i eksempelet over.

If-setninger

En `if`-test ser slik ut i Matlab og C++:

```
b = 2;
if b > 2
        disp('B is greater than 2');
else
        disp('B is less than or equal to 2');
end
```

(a) Matlab

```
int b = 2;
if (b > 2) {
        std::cout << "B is greater than 2" << std::endl;
} else {
        std::cout << "B is less than or equal to 2" << std::endl;
}
```

(b) C++

Figur 4: If-tester i Matlab og C++

En viktig forskjell mellom `if`-tester i Matlab og C++ er at Matlab ikke krever parenteser rundt det som skal testes. C++ krever dessuten krøllparenteser rundt kodeblokken som kjøres dersom `if`-testen er sann, og tilsvarende for `else`.[1]

[1] Dersom kodeblokken kun er én linje lang kan krøllparentesene droppes. Imidlertid er dette en vanlig kilde til feil: hvis man legger flere linjer til kodeblokken, vil det uten krøllparenteser fortsatt kun være den første linjen som «omfattes» av testen, og de andre linjene vil bli kjørt uansett.

For-løkker

En enkel `for`-løkke som kjører fra 1 til 10 ser slik ut i Matlab og C++:

```matlab
for i = 1:10
    disp(i);
end
```

(a) Matlab

```cpp
for (int i = 1; i <= 10; i++) {
    std::cout << i << std::endl;
}
```

(b) C++

Figur 5: Løkker i Matlab og C++

`for` ser litt annerledes ut, men vi finner igjen 1 og 10 som grenser også her. At vi har en steglengde på 1 er litt mindre intuitivt. Dette kommer av den siste delen av argumentet til `for`-løkken, `i++`, som tilsvarer `i = i + 1` eller i ord «øk `i` med 1». I eksempelet under brukes steglengde 2 i stedet, noe som fører til at kun oddetallene skrives ut:

```matlab
for i = 1:2:10
    disp(i)
end
```

(a) Matlab

```cpp
for (int i = 1; i <= 10; i = i + 2) {
    std::cout << i << std::endl;
}
```

(b) C++

Figur 6: Løkker med steglengde i Matlab og C++

Uttrykk på formen `i = i + x` skrives ofte som `i += x` i C++. I vårt tilfelle kunne vi altså i stedet skrevet `i += 2`. Legg også merke til at vi uttrykker lengden på for-løkka vår som en sammenligning, `i <= 10`. Dette fungerer slik at `for`-løkken fortsetter så lenge denne sammenligningen er sann. For ordens skyld er det i C++ normalt å starte løkkene på 0 dermed og bruke $<$ i stedet for $<=$, blant annet fordi tabeller i C++ har første indeks på 0, og ikke på 1 som i Matlab.

While-løkker

```matlab
i = 1;
while i < 1000
        i = i * 2;
        disp(i);
end
```

(a) Matlab

```cpp
int i = 1;
while (i < 1000) {
        i = i*2;
        std::cout << i << std::endl;
}
```

(b) C++

Figur 7: `while`-løkker i Matlab og C++

Igjen er det kun mindre forskjeller i syntaks mellom en `while`-løkke i Matlab og en i C++. Det er likevel greit å merke seg at C++ her også krever at det er parenteser rundt utrykket som testes og at i stedet for å markere slutten på `while`-løkken med `end` bruker C++ krøllparenteser.

Tabeller

```
t = zeros(1,10);
for i = 1:10
    t(i) = i;
    disp(t(i));
end
```

(a) Matlab

```
int t[10];
for (int i = 0; i < 10; i++) {
        t[i] = i;
        std::cout << t[i] << std::endl;
}
```

(b) C++

Figur 8: Tabeller i Matlab og C++

Tabeller opprettes litt annerledes måte i Matlab enn i C++. For å opprette en tabell i C++ er vi (inntil videre) avhengig av å vite den endelige størrelsen før programmet kompileres, slik at det kan settes av nok minne. Når tabellen er opprettet vil den oppføre seg på nesten tilsvarende måte som i Matlab. Det er imidlertid viktig å bite seg merke i forskjellene her:

- Tabell-indeksene starter på 0 i C++, i motsetning til 1 i Matlab. Dette betyr at en tabell på 10 elementer har indeksene $0, 1, \ldots, 9$. Legg deg dette på minne, da det fort kan være litt uvant.
- Vi bruker `[]` og ikke `()` til å indeksere med i C++.
- Tabellen har ikke noen veldefinert startverdi. Hvis du ikke tilordner en verdi, vil det ligge «tilfeldige» verdier i cellene til tabellen, definert ut ifra hva som tilfeldigvis lå i minnet der tabellen ble opprettet. Dette er ofte, men ikke alltid 0, så ting kan se pent ut når man kjører én gang, men bli feil neste om man ikke tilordner verdier. Alternativt kan dette føre til at programmet regelrett krasjer.
- Tabeller kan i utgangspunktet ikke endre størrelse eller vokse. Imidlertid skal vi i løpet av faget lære om løsninger for å komme rundt denne begrensningen.

Funksjoner

```
function [ result ] = getFour(seed)
    result = 4;
end
```

(a) Matlab

```
int getFour(int seed) {
    return 4;
}
```

(b) C++

Figur 9: Funksjoner i Matlab og C++

I Matlab defineres funksjoner ved å skrive `function returverdi = funksjonsnavn(parameter)`. Dette er ulikt måten vi definerer funksjoner på i C++, der vi også må ha med *typen til funksjonens returverdi*. Returverdien til funksjoner i C++ er i motsetning til i Matlab ikke navngitt. I stedet bruker man `return` til å eksplisitt returnere en gitt variabel fra funksjonen og avslutte kjøring av funksjonen. Som når variabler vanligvis defineres må funksjonens argumenter defineres med typer. I

dette tilfellet ser vi at funksjonen tar inn et heltall, `int`. Til sist bemerker vi at funksjoner i C++ ikke trenger å lagres i en fil med samme navn som funksjonen. Man har vanligvis flere funksjoner i samme fil, særlig hvis mange av funksjonene er korte eller henger sammen.

Potens-operatoren

I Matlab bruker vi operatoren `^` for å opphøye et tall. I C++ finnes ikke denne, men vi har to andre alternativer. For enkle uttrykk, der eksponenten er et lite heltall, kan vi eksplisitt multiplisere tallet vi vil opphøye med seg selv, f.eks. $x^3 = x * x * x$. Alternativt kan vi inkludere biblioteket `cmath` og bruke funksjonen `std::pow`.

```
#include <iostream>
#include <cmath>

int main() {
        int fourSquared = std::pow(4,2);
        int fourCubed = std::pow(4,3);
        std::cout << "4^2: " << fourSquared
        << " 4^3: " << fourCubed << std::endl;
        return 0;
}
```

Figur 10: Eksempel på opphøying ved hjelp av `std::pow`

En ting å merke seg fra dette eksempelet er at uttrykket med `std::cout` går over to linjer. Dersom en linje ikke ender med semikolon, vil kompilatoren forsøke å tolke neste linje som en fortsettelse av det aktuelle uttrykket. Ofte separerer man logisk separate deler av et langt uttrykk på denne måten, slik at koden blir lettere å lese og linjene ikke blir for lange.

1 Kodeforståelse: oversett til Matlab (10%)

a) Oversett følgende kodesnutt til Matlab.

```
bool isFibonacciNumber(int n){
    int a = 0;
    int b = 1;
    while (b < n){
        int temp = b;
        b = a + b;
        a = temp;
    }
    if (b == n){
        return true;
    }
    else{
        return false;
    }
}
```

Figur 11: C++-kode

2 Oversett fra Matlab til C++ (90%)

Oversett følgende kodesnutter til C++, og sjekk at de både kompilerer og kjører i ditt IDE.

a) Største av to tall

```
function [ largest ] = maxOfTwo( a, b )
    if a > b
        disp('A is greater than B');
        largest = a;
    else
        disp('B is greater than A');
        largest = b;
    end
end
```

Figur 12: Matlab-kode

b) Fibonaccirekker

```
function [ result ] = fibonacci( n )
    a = 0;
    b = 1;
    disp('Fibonacci numbers:')
    for x = 1:n
        fprintf('%d: %d\n', x, b)
        temp = b;
        b = a + b;
        a = temp;
    end
    disp('----');
    result = b;
end
```

Figur 13: Matlab-kode

c) Sum av kvadrerte tall

```
function [ totalSum ] = squareNumberSum( n )
    totalSum = 0;
    for i = 1:n
        totalSum = totalSum + i^2;
        disp(i^2);
    end
    disp(totalSum)
end
```

Figur 14: Matlab-kode

d) Trekanttall

```
function triangleNumbersBelow( n )
    acc = 1;
    num = 2;
    fprintf('Triangle numbers below %d: ', n);
    while acc + num < n
        acc = acc + num;
        num = num + 1;
        fprintf('%d ',acc);
    end
    fprintf('\n');
end
```

Figur 15: Matlab-kode

```
function [ truthValue ] = isTriangleNumber( number )
    acc = 1;
    while number > 0
        number = number - acc;
        acc = acc + 1;
    end
    if number == 0
        truthValue = true;
    else
        truthValue = false;
    end
end
```

Figur 16: Matlab-kode

e) Primtall 1

```
function [ primeness ] = isPrime( n )
    primeness = true;
    for i = 2:(n-1)
        if mod(n,i) == 0
            primeness = false;
            break;
        end
    end
end
```

Figur 17: Matlab-kode

f) Primtall 2

```
function naivePrimeNumberSearch( n )
    for number = 2:(n-1)
        if isPrime(number) == true
            fprintf('%d is a prime\n', number);
        end
    end
end
```

Figur 18: Matlab-kode

g) Største fellesnevner

```
function [ greatestDivisor ] = findGreatestDivisor( n )
    for divisor = (n-1):-1:0
        if mod(n,divisor) == 0
            greatestDivisor = divisor
            break;
        end
    end
end
```

Figur 19: Matlab-kode

h) Telling med lister

```
function [ noReturn ] = compareListOfNumbers( l )
    r = zeros(1,3);
    for i = 1:length(l)
        if l(i) < 0
            r(1) = r(1) + 1;
        elseif l(i) == 0
            r(2) = r(2) + 1;
        else
            r(3) = r(3) + 1;
        end
    end
    fprintf('%d numbers were below zero\n', r(1));
    fprintf('%d numbers were zero\n', r(2));
    fprintf('%d numbers were greater than zero\n', r(3));
end
```

Figur 20: Matlab-kode