

# TDT4102

## Prosedyre- og objektorientert programmering

The image features a large, stylized 'C++' logo in the center. Behind the logo is a snippet of C++ code, tilted at an angle. The code includes a header, namespace declaration, and a main function that prints 'Hello World!'.

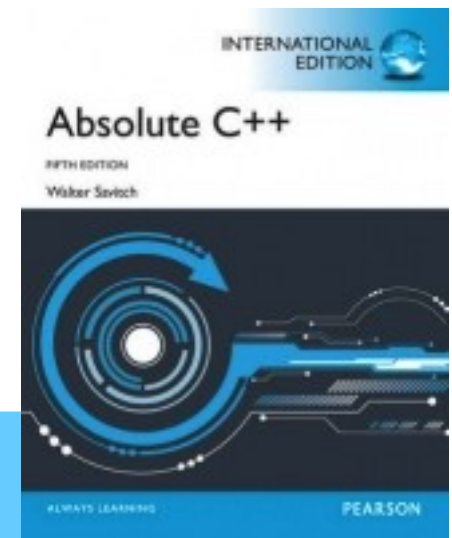
```
#include <iostream>
using namespace std;
int main() {
    cout << "Hello World!\n";
    return 0;
}
```

## Funksjoner

# Innhold

- Funksjoner
- Scope
- Pekere (og referanser)
- Overlagring
- Litt om testing

Kap. 4, 5, 10.1





# Funksjoner: litt repetisjon

- Funksjoner er viktig i programmering
  - Bruk funksjoner, tenk funksjoner, lag funksjoner
- Funksjoner er ”byggeklosser”
  - en logisk enhet av funksjonalitet
  - forenkler problemløsning – løse mange små overkommelige problemer i stedet for et stort sammensatt problem
  - gjør koding lettere - skrive, lese, teste, vedlikeholde
  - gjenbruk er et viktig prinsipp i programmering



# Bruke funksjoner i et bibliotek

- C++ har mange funksjoner som vi skal bruke
  - Organisert i forskjellige bibliotek
  - For input/output, matematiske funksjoner, strengfunksjoner, etc.
  - Vanlig feil å glemme å include-statement for bibliotek
  - Noen header-filer inkluderes automatisk av kompilatoren
- Må bruke: `#include <bibliotek>`
  - Gjør at kompilatoren inkluderer "spesifikasjonene" av funksjoner og typer i biblioteket (**header-fil**)
- `#include <bibliotek>` brukes for header-filer i systemstien, mens `#include "header.h"` brukes for header-filer i lokale kataloger (som ligger i prosjektet ditt)

# Funksjon for å generere tilfeldige tall



```
#include <cstdlib>
```

- I øvingene vil du ha bruk for tilfeldige tall
  - også viktig i mange vanlige programmer
- `int rand();`
  - tar ingen argumenter, returnerer heltall fra og med 0 til og med konstanten `RAND_MAX`
- Skalering?
  - Hvordan få tall mellom andre verdier?
  - `rand() % 10`
- Forskyve verdiene?
  - `(rand() % 10) + 1`

# Tilfeldig? – ikke helt....



- Pseudo-tilfeldige tall
  - gjentatte kall til rand() resulterer i en sekvens av “tilfeldige” tall
  - algoritmisk produsert
- Kan bruke “seed” til å endre denne sekvensen:

```
void srand(int seed);
```

- void funksjon som tar en int som argument
- kan bruke en hvilken som helst verdi, f.eks. system-tiden
- biblioteket <ctime> inneholder en time-funksjon

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

# Eksempel - med litt repetisjon

```
#include <iostream>
#include <cstdlib>
#include <ctime>

using namespace std;

int main(){

    srand(static_cast<unsigned int>(time(NULL)));

    int secret = (rand() % 10) + 1;
    int guess = 0;
    int count = 0;

    while (guess != secret){
        cout << "Gjett et tall mellom 1 og 10: ";
        cin >> guess;
        count ++;
        if (guess > secret){
            cout << "Tallet er for stort" << endl;
        }else if(guess < secret){
            cout << "Tallet er for lite" << endl;
        }else{
            cout << "Du har gjettest riktig (" << count << ")" << endl;
        }
    }
}
```



# Egendefinerte funksjoner

- Å skrive egne funksjoner er en viktig del av å konstruere et program!
- Funksjoner er byggeklosser
- Gjør koden ryddig, enklere å teste og finne feil, lettere å samarbeide om
- Dine funksjoner kan ligge i
  - samme fil som main()
  - eller i egne filer





# Deklarering, implementering og bruk

- Deklarering av funksjonen

- Funksjonsprototypen
- Så kompilatoren kan sjekke at funksjonen brukes riktig

Vi skiller med andre ord mellom deklarasjon og implementasjon

- Funksjonens implementasjon

- Den faktiske koden som utføres

- Bruk av funksjonen

- Aktivere funksjonen
- Kalle funksjonen fra main() eller fra andre funksjoner

# Eksempel

- Vi kan lage oss en funksjon som ber om input fra bruker

```
//Ber bruker om tall mellom min og max
int getGuess(int min, int max){
    int temp = 0;
    do{
        cout << "Gjett et tall mellom ";
        cout << min << " og " << max << ":";
        cin >> temp;
    }while(temp < min || temp > max);
    return temp;
}
```



# Funksjonsprototypen

- Er en informativ deklarasjon for kompilatoren
- Forteller kompilatoren hvordan funksjonen brukes:
  - hvilke typer som er lovlige som input og i hvilke uttrykk vi kan bruke funksjonen
- SYNTAKS:
 

```
<returtype> funksjonsNavn (<parameter-liste>);
```

```
int getGuess(int min, int max);
```
- Må stå før du bruker funksjonen i koden:
  - I første omgang plasserer vi denne over main()-linjen
  - Kalles også funksjons**prototype**
  - Vi kan utelate variabelnavnene i prototypens parameterliste

# Parameter og returverdi

```
int getGuess(int min, int max);
```

- Parameterlista definerer hvilke verdier du kan sende til funksjonen
  - Datatyper, antall verdier
  - Bruk må samsvare med deklarasjonen
    - Datatype, antall, rekkefølge
- Returtypen definerer hvilken datatype funksjonen returnerer
  - Kun en enkelt verdi kan returneres
  - Returverdi må stemme med deklart type

```
//Ber bruker om tall mellom min og max
int getGuess(int min, int max);
```

```
//Skriver ut informasjon
void printGuess(int guess, int secret, int count);
```

```
int main(){
    srand(static_cast<unsigned int>(time(NULL)));
    int secret = (rand() % 10) + 1;
    int guess = 0;
    int count = 0;

    while (guess != secret){
        guess = getGuess(1, 10);
        count++;
        printGuess(guess, secret, count);
    }
}
```

```
void printGuess(int guess, int secret, int count){
    if (guess > secret){
        cout << "Tallet er for stort" << endl;
    }else if(guess < secret){
        cout << "Tallet er for lite" << endl;
    }else{
        cout << "Du har gjettet riktig (" << count << ") " << endl;
    }
}
```

```
int getGuess(int min, int max){
    int temp = 0;
    do{
        cout << "Gjett et tall mellom ";
        cout << min << " og " << max << ": ";
        cin >> temp;
    }while(temp < min || temp > max);
    return temp;
}
```

Rekkefølgen hvis alt er i samme .cpp fil

Prototypene først

Deretter kommer koden som bruker funksjonene

Selve implementasjonene kan komme til slutt

# Hvordan lagrer jeg funksjoner i egne filer?



- Funksjoner kan også ligge i egne filer
  - koden for litt større program er bestandig fordelt over flere filer
  - praktisk å ha flere filer med mindre kode i hver
- Lagre deklarasjonene i "filnavn.h" fil
- Lagre implementasjonene i egen "filnavn.cpp" fil
- Bruk `#include "filnavn.h"` for å bruke funksjonene i et program



# Scope

- En variabels **scope** er den delen av et program hvor variabelen kan bli brukt
  - tilordnes verdier, bruke verdien
- Scope **starter** der en variabel blir deklarerert og slutter der blokken den er definert i **slutter**
- Når en blokk avsluttes vil også variablene som var deklarerert i blokka "forsvinne"
- En funksjonsimplementasjon er en egen blokk

```
{
    int x = 1;
    {
        int y = x;
    }
}
```

# Variabler med samme navn

- Identifikatoren (navnet til en variabel) er **assosiert med scope**
- Vi kan derfor i koden vår ha flere variabler med samme navn så lenge de har forskjellig scope

```
int i = 1;
int sum = 0;

for (int i = 0; i < 10; i ++){
    sum += i;
}

for (int i = 20; i < 50; i ++){
    sum += i;
}
```

- NB! Det er også tillatt å bruke navn som finnes i foreldre-blokken
  - men da vil den nye variabelen "overskygge" variabelen i foreldre-blokka
  - generelt en dårlig praksis å gjøre dette, men greit å vite siden det er en kilde til feil





# Lokale og globale variabler

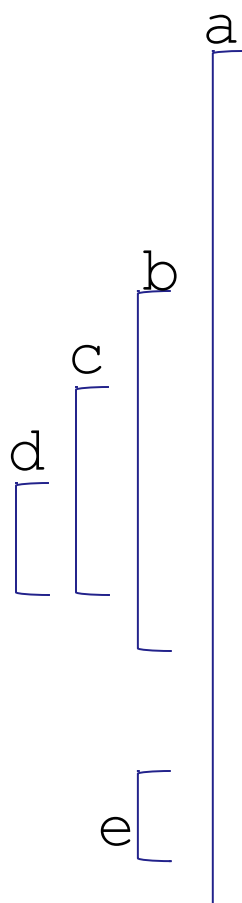
- **Lokale** variabler
  - Variabler som deklarerer inne i funksjoner (inkl. main) eller underblokker av funksjonene
- **Globale** variabler
  - Variabler deklarerert utenfor alle funksjonene (også utenfor main)
  - Disse vil bli tilgjengelig for alle funksjoner og har default initialisering (f.eks til 0 for int og double)
- *Av og til er globale variabler OK....*
  - *for eksempel for konstanter*
  - *men god praksis tilsier å unngå dette for variabler som kan endres*

# Eksempel

```
#include <iostream>
using namespace std;
int main() {
    cout << "Hello World!\n";
    return 0;
}
```

variablenes  
forskjellige  
scope

vi kan bruke  
debugging for  
å følge med  
variablenes verdier



```
#include <iostream>
using namespace std;

int a = 0;
void akkumuler(int e);

int main() {

    int b = 1;

    for (int c = 0; c < 4; c++)
    {
        int d = b + c;
        akkumuler(d);
    }
    cout <<  a << endl;

    void akkumuler(int e) {
        a += e;
    }
}
```

# Funksjonskall kopierer verdien til argumentene

```
#include <iostream>
using namespace std;

void akkumuler(int);

int main(){
    int sum = 0;
    for (int i = 0; i < 10; i++){
        akkumuler(sum, i);
    }
    cout << "Resultat = " << sum << endl;
}

void akkumuler(int sum, int i){
    sum += i;
}
```

*Hva skjer her?*

*Parametervariabelen sum er her en lokal variabel som forsvinner når funksjonen avslutter.*



# static variabler

- Parametre og andre variabler som deklarerer og brukes i et funksjonskall eksisterer kun så lenge funksjonen er aktiv
  - neste gang funksjonen kalles er det "nye" variabler som brukes
- Resultatet er at funksjoner ikke kan ta vare på verdier fra kall til kall
- Men av og til har vi bruk for å ta vare på verdier fra kall til kall
  - ikke noe som skal brukes til vanlig!
  - men nyttig for å løse spesielle problemer
- Variabler deklarert som static vil initialiseres en gang og deretter beholde sin verdi
  - static er et nøkkelord som vi setter foran datatypen
  - slike nøkkelord kalles generelt for modifikatorer

```
int akkumuler(int i){
    static int akk = 0;
    akk += i;
    return akk;
}
```

# Pekere

- I C og C++ finner du en egen datatype som kalles for pekere
- Pekere er variabler som peker til andre variabler
- Er en form for indirekte adressering som er nødvendige i mange sammenhenger
  - Men som mange andre prog. språk velger å skjule
  - Lagrer adressen til en annen variabel
- En peker deklarerer til å peke til en variabel av en spesifikk datatype

# peker \*syntaks

- Stjernetegnet `*` brukes når du deklarerer pekere

```
int hoyde = 5;
int bredde = 4;
int *hoydePeker = &hoyde;
int *breddePeker = &bredde;
```

- Her initialiserer vi med adresser, men vi kan også ha pekere som ikke peker til noe

```
int *pekerTilIngenting = nullptr;
```

- Stjernetegnet brukes også når vi skal lese/skrive verdiene det pekes til

```
cout << *hoydePeker << endl;
```

- da heter det derefereringsoperator

# Eksempel

Minne
...
5
8
&hoyde
&bredde
...

```
int hoyde = 5;
int bredde = 8;
int *hoydePeker = &hoyde;
int *breddePeker = &bredde;
```

Vi legger til følgende kode:

```
hoydePeker = breddePeker;
*hoydePeker = 4;
```

Hvilket tall er endret til 4?  
Hvordan peker pekerne nå?

Minne
...
5
<b>4</b>
&bredde
&bredde
...



# Parameter i C++

- Call-by-value

- Funksjonen mottar en kopi av argumentverdien
- Vi kan gjøre hva vi vil med denne i en funksjon, uten at “originalen” endres
- Det er denne mekanismen som brukes når parameterne er deklarert på vanlig måte, f.eks. `int anyfunction(int) ;`

- Call-by-pointer (peker)

- Vi sender en adresse som argumentverdi
- Endres variabelverdien i en funksjon så endrer vi også “originalen”
- Deklareres med: `int anyfunction(int*) ;`





# Peker som parameter

- Spesifiseres med tegnet “\*” etter typedefinisjon i parameterlista (parameter deklarert som peker)
- Brukes når vi vil at funksjonen skal kunne endre på argumentvariabelen den ble kalt med
- Brukes når vi trenger
  - Funksjoner som skal gi mer enn en verdi som resultat
  - Når vi vil unngå unødvendig kopiering ved funksjonskall (mest aktuelt for datatyper vi skal lære om seinere)

# Oppgave: swap

- Hvordan "swappe" to verdier?
- Koden for hvordan to variabler bytter verdi.

# Verdi vs. Peker i funksjonskall



```
void mySwap(int a, int b){
    int temp = a;
    a = b;
    b = temp;
}
```

```
int main(){
    int a = 2;
    int b = 4;
    mySwap(a, b);
}
```

Har ingen effekt på  
variablene i main()

Løsningen er å  
bruke pekere

```
void mySwap(int *a, int *b){
    int temp = *a;
    *a = *b;
    *b = temp;
}
```

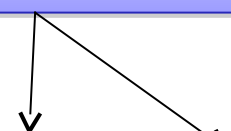
```
int main(){
    int a = 2;
    int b = 4;
    mySwap(&a, &b);
}
```

# Referanser

## (i stedet for å bruke pekere)

- Pekere kan være vanskelig å holde styr på
- C++ har en forenklet mekanisme som heter referanser og "call-by-reference"
- &-tegn i dekl. av parameter
- Implisitt dereferering

Her er det referanser som brukes



```
void mySwap(int &a, int &b){  
    int temp = a;  
    a = b;  
    b = temp;  
}  
  
int main(){  
    int a = 2;  
    int b = 4;  
    mySwap(a, b);  
}
```



# Call-By-Reference

- Det vi sender er en referanse til kallende kodes argument
  - Er adressen til argumentets minnelokasjon
  - Brukes (vanligvis) når vi faktisk ønsker at funksjonen skal endre variablene som er brukt som argument
- Men av og til ønsker vi å bruke referanse-parameter, uten at variablene skal endres
  - for eksempel for å lage høyeffektive programmer som bruker minst mulig minne og ressurser på å kopiere verdier ved funksjonskall
- Kan "beskytte" parameter-referanser ved å bruke modifikatoren **const**
  - "read only" `int anyfunction(const int &a);`
  - vil gi kompileringsfeil hvis vi skriver kode i `anyfunction` som endrer verdien av variabelen `a`

# Parameterlista revisited



- Vi kan godt ha en parameterliste som bruker flere mekanismer

```
int anotherFunction(int a, int &b, int *c);
```

- Første argument sendes som verdi
- Andre sendes som referanse
- Tredje sendes som peker

# Overlagring

## eller på engelsk: overloading



- Navnet på en funksjon er bare ett av flere elementer som identifiserer en funksjon!
  - Funksjonens **signatur** er navn og parameterliste
  - Inkluderer ikke const og adresse-av-operatoren (&)
- Det er lov å ha flere funksjoner med samme navn
  - så lenge parameterlisten er forskjellig!
- Dette kalles **overlagring** og er ganske nyttig!
- Gir oss mulighet til å ha varianter av samme funksjonalitet som kan utføres på forskjellige data
  - Samme navn, men forskjellig parameterliste
  - Returtype kan også være forskjellig, men returtype brukes ikke for å avgjøre hvilken funksjon som skal kalle



# Overlagring: eksempel

- Funksjon som beregner gjennomsnitt og returnerer en double verdi
- Siden overlagring er mulig kan vi lage én funksjon som tar 2 argumenter og en funksjon som tar 3 argument

```
double avarage(double a, double b){
    return (a + b) / 2;
}

double avarage(double a, double b, double c){
    return (a + b + c) / 3;
}
```





# Hvilken funksjon kalles?

- Avhenger av funksjonskallet
  - kompilatoren søker etter navn + parameterliste som matcher

```
int main(){
    ★ double x = avarage(2.0, 5.0);
    ★ double y = avarage(1.0, 3.0, 4.0);
}

double avarage(double a, double b){
    return (a + b) / 2;
}

double avarage(double a, double b, double c){
    return (a + b + c) / 3;
}
```



# Overlagring forts.

- Bruk det KUN for like oppgaver!
  - samme funksjonalitet, forskjellig input
- Regler for matching
  - 1: finn eksakt signatur
  - 2: finn kompatibel signatur som kan brukes med implisitt casting UTEN tap (eks: `int -> double`)
  - 3: finn kompatibel signatur som kan brukes med implisitt casting MED tap (eks: `double -> int`)
  - Hvis umulig å avgjøre → kompileringsfeil



# Default argumenter

- I C++ er det mulig å spesifisere default verdier for argumentene til en funksjon
- Spesifiseres i funksjonsprototypen
  - men kan utelates i implementasjonsdelen

```
double beregnVolum(double h, double b = 1.0, double l = 1.0);
```

- Mulige kall:
  - beregnVolum(2, 4, 6);
  - beregnVolum(3, 5);
  - beregnVolum(7);

NB! Det er kun de siste parametrene som kan ha default verdi hvis det også er parametre uten defaultverdier i lista!!!!



# Testing

- Det er fort gjort å gjøre feil ved koding
  - Mange feil er av typen syntaksfeil og gir kode som ikke kompilerer
  - Men det kan være mange feil som fører til feil "oppførsel"
  - Programmet kjører, men oppfører seg feil
- Testing er en sentral del av moderne programutvikling



# Forskjellige måter å teste på

- Enkel testing mens du programmerer:
  - Prøv ut programmet ditt med forskjellige verdier
  - Skriv ut til cout (eller cerr) verdier av variabler før og etter funksjonskall etc.
  - Bruke debugger; sjekke hvordan variabler endres og hvordan funksjoner kalles
- Vi kan også teste mer systematisk :
  - Vi tester hver “enhet” vi lager separat (i praksis testing av funksjonene)
  - Vi tester på betingelser før kall og etter kall til en funksjon



# Regler og betingelser

- Definere **regler** for dine funksjoner
  - Hva er forventet resultat for gitte verdier?
  - Hva er forventet resultat for “grenseverdier” eller spesialtilfeller?
- Reglene kan testes med **betingelser**
  - Hvilke betingelser må gjelde før en funksjon kalles
  - Hvilke betingelser skal gjelde etter at den er kalt
- Testing basert på regler og betingelser er en metode for systematisk testing



# assert

- I C++ kan vi bruke en assert makro for å teste koden vår
- En assertion (påstand) er et statement som enten er true eller false
- Brukes for å sjekke betingelser (for alle reglene vi har)
  - Preconditions & Postconditions -> før og etter kall til funksjonen
- Syntaks: **assert(<assert\_betingelse>);**
  - Ingen returverdi
  - Evaluerer assert\_betingelsen
  - Avslutter programmet hvis false, fortsetter hvis true
- Predefined i biblioteket <cassert>
  - Makroer brukes likt med funksjoner
  - Men ved kompilering (preprosessering) vil makroen erstattes med makrokoden



# Bruk av assert makroen

- assert bruker du som en vanlig funksjon
- men assert-statements kan “slåes” av og på med preprosessor direktivet **#NDEBUG**

– #NDEBUG  
#include <cassert>



assert makroene er AV og programmet kompiles uten at disse er med

– //#NDEBUG  
#include <cassert>



assert makroene er PÅ og programmet vil avslutte hvis et assert-statement => false

utelater vi #NDEBUG helt vil koden kompiles MED assert





# Debugging

- Vi har ofte bruk for å kunne undersøke programmet vårt trinn for trinn
  - for å forstå hvordan koden fungerer
  - for å finne feil
- Debugging er en nyttig funksjonalitet som lar oss gjøre dette!
  - I VS: sett breakpoint ved å klikke i venstremargen på kildekoden
  - Start programmet i debug-modus
  - Bruk F10 og F11 for å gå trinnvis gjennom programmet
- Mer om dette på øvingsforelesing!



# Oppsummering

- *Bruke funksjoner og litt repetisjon*
- *Egendefinerte funksjoner*
  - *deklarasjon og implementasjon*
  - *parameter/argumenter, returverdier*
- *Scope*
  - *synlighet av variabler*
  - *lokale og globale variabler*
- *Pekere (og referanser)*
- *Overlagring*
- Neste forelesing er om tabeller (arrays)
- Sjekk kap. 5