TDT4102
Prosedyre- og objektorientert programmering

Funksjoner m.m.

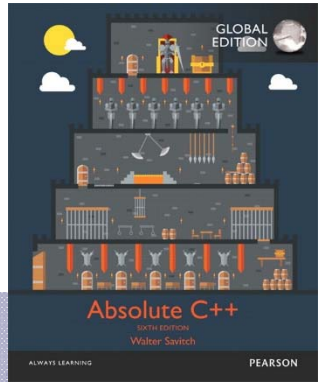
Forrige uke

- Mye administrativ info. om faget
  - Les slides!
    - Bl.a. henvendelser, mailing-lister
    - Ca. 800 studenter: Stud.ass → und.ass → fag-ans
  - Les alt på Its learning
  - Vi får stadig nye påmeldinger
    - Vær litt tålmodige med oss
  - Bruk piazza om nødvendig
- Introduserte Kattis
  - La ut kode for problemet «Oddities»

Innhold i dag / denne uke

- Eksempel: Meny – switch
- Funksjoner, parametere
- Tilfeldige tall (rand, srand)
- Scope (Gyldighetsområde)
- Pekere (og referanser)
- Overlagring
- Litt om testing

Kap. 3-4



Eksempel, enkel meny med **switch**

```
1   // TestOvingMal.cpp
2   #include <iostream>
3   using namespace std;
4
5   void oppgave1() { /* empty */ }
6
7   void oppgave2a() { ... }
10  void oppgave2b() { ... }
13  void oppgave2c() { ... }
16  void oppgave2() { ... }
41
42  void oppgave3() { ... }
45
46      void main() { ... }
53
```

Eksempel, enkel meny med **switch**

```
13  void oppgave2c() {
14      cout << "... tester oppgave 2c\n";
15      // din kode her for 2c
16  }
17  void oppgave2() {
18      cout << "Test av oppgave 2 starter" << endl;
19      bool finished = false;
20      char c = ' ';
21      while (!finished) {
22          cout << "Tast a for deloppg. a, b for\n";
23          cout << "deloppgave b osv., q for å avslutte\n";
24          cin >> c;
25          switch (c) { ... }
41      }
42  }
```

```
21      while (!finished) {
22          cout << "Tast a for deloppg. a, b for\n";
23          cout << "deloppgave b osv., q for å avslutte\n";
24          cin >> c;
25          switch (c) {
26          case 'a':
27              oppgave2a();
28              break;
29          case 'b':
30              oppgave2b();
31              break;
32          case 'c':
33              oppgave2c();
34              break;
35          case 'q':
36              finished = true;
37              break;
38          default :
39              cout << "*** ugyldig valg ***\n";
40          }
41      }
42  }
```

Eksempel, enkel meny med **switch**

Parameter og returverdi

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

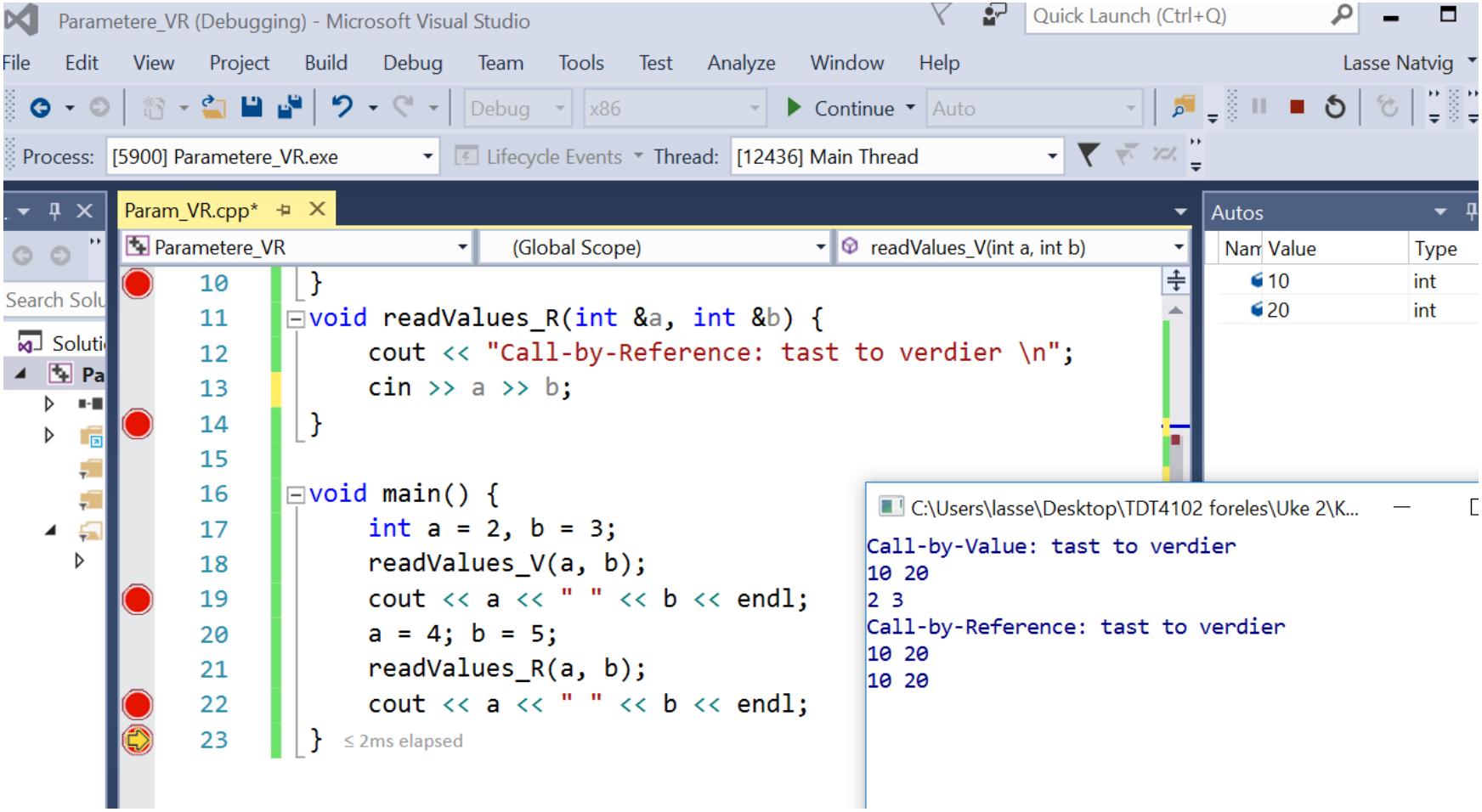
- Parameterlista definerer hvilke verdier du kan sende til funksjonen
  - Datatyper, antall verdier
  - Bruk må samsvare med deklarasjonen
    - Datatype, antall, rekkefølge
  - Parametere i dette eksempel er “Call-by-value”
    - “Verdi sendes inn, men returneres ikke”
- Returtypen definerer hvilken datatype funksjonen returnerer
  - Kun en enkelt verdi kan returneres
  - Returverdi må stemme med deklarert type

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

```
int getValues (int &a, int &b);
```

Call-by-Value vs., call-by-Reference, eksempel med debugger



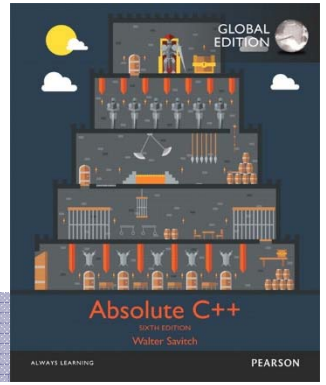
Bruke funksjoner i et bibliotek

- C++ har mange funksjoner som vi skal bruke
  - Organisert i forskjellige bibliotek
  - input/output, matematiske funksjoner, tekststrenghåndtering etc.
- 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  // området [0..9]`
- Forskyve verdiene?
  - `(rand() % 10) + 1 // området [1..10]`



Side 134 – 141, AP-13

Tilfeldig? – ikke helt....

- Pseudo-tilfeldige tall
  - gjentatte kall til rand() resulterer i en sekvens av "tilfeldige" tall
  - algoritmisk produsert
- Kan bruke "seed" (frø) 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));
```

- Seed, fast verdi, fordel = …
  - (mer) deterministisk program-oppførsel
- Seed, ukjent verdi, fordel = …
  - (mer) varierende program-oppførsel

Eksempel - med litt repetisjon

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

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

int main() {
    setlocale(LC_ALL, "Norwegian");
    // ...
    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);
    }
}
```

Eksempel

- be om input fra bruker

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

Eksempel

```
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 (på " << count << " forsøk)" << endl;
    }
}
```

```
//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

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

- Scope starter der en variabel blir deklarert og slutter der blokken den er definert i slutter
- Når en blokk avsluttes vil også variablene som var deklarert i blokka ”forsvinne“
- En funksjonsimplementasjon er en egen blokk

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 deklareres inne i funksjoner (inkl. main) eller underblokker av funksjonene
- Globale variabler
  - Variabler deklarert 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

variablenes forskjellige scope

vi kan bruke debugging for å følge med variablenes verdier

a
b
c
d
e

```
#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 sum, int i);

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;
}
```

Dv. Funksjonen virker IKKE slik den var tiltenkt

Hva skjer her?

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

static variabler

- Parametre og andre variabler som deklareres 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

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

- 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

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 deklareres 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;
```

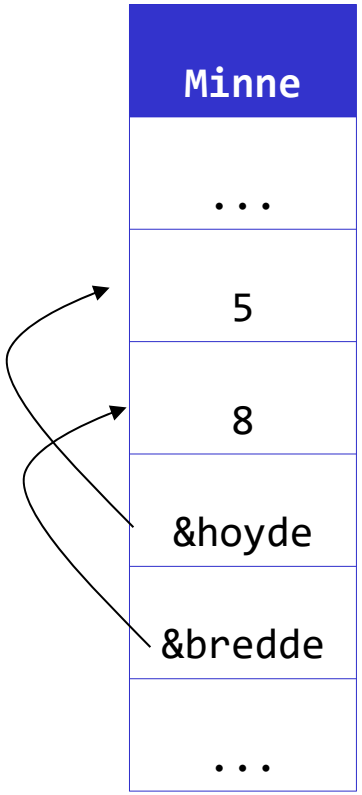
- Stjernetegnet brukes også når vi skal lese/skrive verdiene det pekes til

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

  - da heter det dereferingsoperator

Variable allokert i minne i samme rekkefølge som i programmet

Eksempel



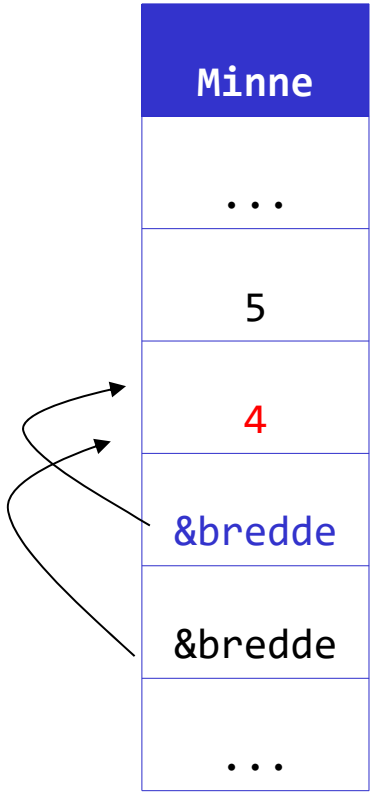

```
int hoyde = 5;
int bredde = 8;
int *hoydePeker = &hoyde;
int *breddePeker = &bredde;
```

Vi legger til følgende kode:

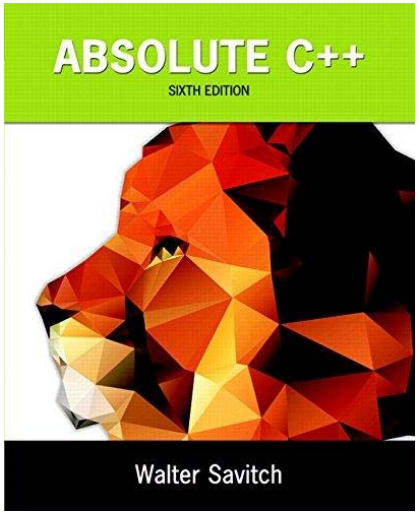
```
hoydePeker = breddePeker;
*hoydePeker = 4;
```

Hvilket tall er endret til 4?

Hvordan peker pekerne nå?



Info fra bokhandel om lærebok, status



Denne er helt OK !

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 "orginalen" 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()

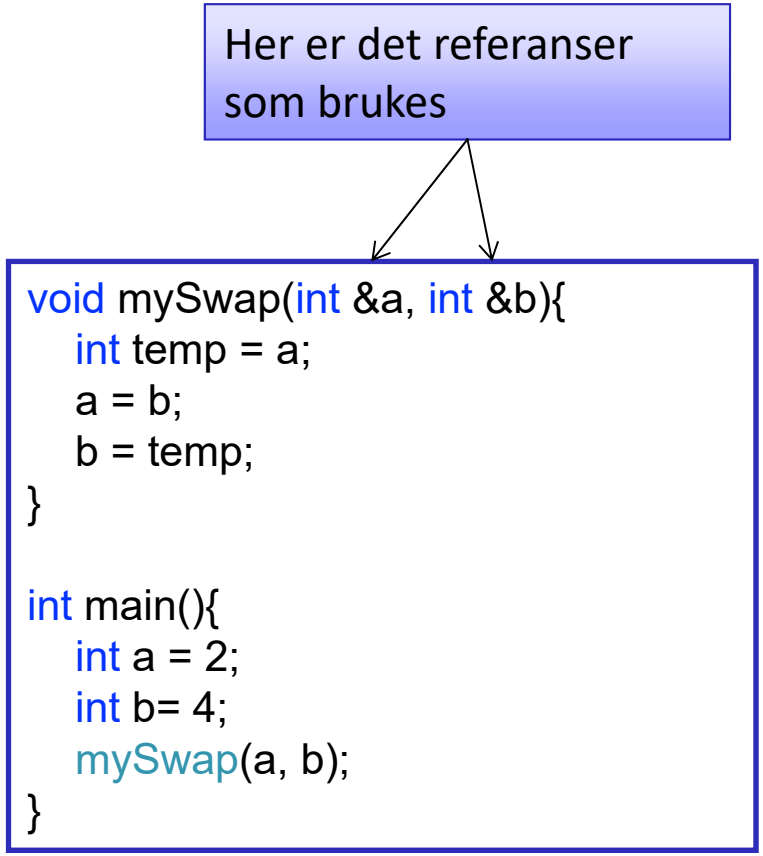
Løsningen er å bruke pekere

```
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



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 kalles

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

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