Øvingsforelesning 5

TDT4102 — Prosedyre- og objektorientert programmering

Haakon K. Garfjell, *und.ass.*
Asta Skirbekk, *und.ass.*

10. februar, 2022

IDI, NTNU

Plan for dagen

Motivasjon for øvingen

enum

Klasser

Klasser vs structs

Motivasjon for øvingen

- Første øvingen hvor dere tar i bruk objektorientert programering!
- Blackjack-oppgaven gir dere frie tøyler og muligheten til å prøve dere som programmerere.
- Sterkt anbefalt å prøve dere selv så mye som mulig.
- Minner veldig om hvordan programmering vil være senere i studiet og arbeidslivet.

Hvordan kode smart?

1. Start med å få en oversikt over problemet som skal løses.
2. Lag en plan for hvordan du skal løse problemet.
3. Del alltid problemet opp i mindre biter (ulike funksjoner).
4. En god tommelfingerregel er at hver funksjon kun skal gjøre én spesifikk ting, med andre ord: hver funksjon skal kun ha én funksjon.
5. Dersom du oppdager at du prøver å løse flere delproblemer i samme funksjon bør du flytte deler av denne koden over i nye funksjoner.
6. **TEST ALLTID KODEN FORTLØPENDE!** får du kompilert? Fungerer koden som du ønsker? Hvis ikke: finn ut hvorfor og rett det opp med en gang. Det er mye lettere å feilsøke 10 kodelinjer enn 100.

Lage egne datatyper

- Hittil har vi brukt innebygde datatyper i C++.
- Vi har brukt noen primitive typer som `int`, `double` og `char`.
- Vi har også brukt noen klasser som `vector` og `string`.
- Nå skal vi lære hvordan man kan lage egne typer!
- Disse **brukerdefinerte typene** gjør at vi kan lage datatyper som passer perfekt til akkurat det vi skulle ønske.

Plan for dagen

Motivasjon for øvingen

enum

Klasser

Klasser vs structs

enum

- `enum` (kommer fra "enumeration") er en veldig enkel, men nyttig datatype.
- En `enum` brukes til å gruppere heltall på en måte slik at de får faste "symboler", egne navn som kan brukes som datatyper.
- Hvert symbol har en korresponderende `int`.
- Syntaks:
  `enum class MyEnum{type0, type1, type2}`.
- **Merk:** Vi kommer til å benytte oss av scoped `enum` som skiller seg noe fra plain `enum`.
- Det er anbefalt å definere `enum` globalt i en .h-fil, slik at den kan brukes av andre funksjoner og klasser.

enum - Eksempel

- Hvordan opprette en **enum**.

```
enum class Animal {cat, dog, horse};
int main() {




}
```

enum - Eksempel

- Hvordan opprette en **enum**.

```
enum class Animal {cat, dog, horse};

int main() {
    Animal pet {Animal::cat};









}
```

enum - Eksempel

- Hvordan opprette en **enum**.

```
enum class Animal {cat, dog, horse};

int main() {
    Animal pet {Animal::cat};
    if (pet == Animal::cat) {
        cout << "My pet is a cat";
    }
    else if (pet == Animal::dog) {
        cout << "My pet is a dog";
    }
    else  {
        cout << "My pet is a horse";
    }
}
```

Enum og heltall

- Hvert enum-symboler tilsvarer et heltall.
- Som standard går tallene fra 0 og oppover.
- Vi kan konvertere mellom heltall og enum-symboler med `static_cast`.

```
enum class Animal {cat, dog, horse};

int main() {
    cout << static_cast<int>(Animal::cat) << endl;
    cout << static_cast<int>(Animal::dog) << endl;
    cout << static_cast<int>(Animal::horse) << endl;

    Animal myAnimal = static_cast<Animal>(0);

}
```

Enum og heltall

- Hvert enum-symboler tilsvarer et heltall.
- Som standard går tallene fra 0 og oppover.
- Vi kan konvertere mellom heltall og enum-elementer med `static_cast`.

```
enum class Animal {cat, dog, horse};

int main() {
    cout << static_cast<int>(Animal::cat) << endl;      // 0
    cout << static_cast<int>(Animal::dog) << endl;      // 1
    cout << static_cast<int>(Animal::horse) << endl;    // 2

    Animal myAnimal = static_cast<Animal>(0);
    // myAnimal = Animal::cat
}
```

Enum og heltall

- Vi kan velge andre heltall

```
enum class Animal {cat = 2, dog, horse = 11};

int main() {
    cout << static_cast<int>(Animal::cat) << endl;
    cout << static_cast<int>(Animal::dog) << endl;
    cout << static_cast<int>(Animal::horse) << endl;

    Animal myAnimal = static_cast<Animal>(2);

}
```

Enum og heltall

- Vi kan velge andre heltall

```
enum class Animal {cat = 2, dog, horse = 11};

int main() {
    cout << static_cast<int>(Animal::cat) << endl;      // 2
    cout << static_cast<int>(Animal::dog) << endl;      // 3
    cout << static_cast<int>(Animal::horse) << endl;    // 11

    Animal myAnimal = static_cast<Animal>(2);
    // myAnimal = Animal::cat
}
```

enum - Eksempel

- Bruk av `enum` i en `switch` ved å konvertere til heltall:

```
enum class Animal {cat, dog, horse};

int main() {
    Animal pet {Animal::cat};
    int intPet {static_cast<int>(pet)};









}
```

enum - Eksempel

- Bruk av `enum` i en `switch` ved å konvertere til heltall:

```
enum class Animal {cat, dog, horse};

int main() {
    Animal pet {Animal::cat};
    int intPet {static_cast<int>(pet)};

    switch(intPet) {
    case 0:
        cout << "My pet is a cat";
        break;
    case 1:
        // etc
    }
}
```

enum - Eksempel

- Man kan bruke vanlige heltall i stedet for **enum**, ved å for eksempel la 0 bety katt, 1 bety hund og 2 bety hest.
- Hvorfor ikke bare bruke **int**?

```
int main() {
    int intPet {0};

    switch(intPet) {
    case 0:
        cout << "My pet is a cat";
        break;
    case 1:
        // etc
    }
}
```

enum - Eksempel

- Med `enum` blir koden mye mer lesbar og tryggere.
- Det er ikke åpenbart i det forrige eksempelet at for eksempel 0 betyr katt.

```
enum class Animal {cat, dog, horse};

int main() {
    Animal pet {Animal::cat};

    switch(pet) {
    case Animal::cat:
        cout << "My pet is a cat";
        break;
    case Animal::dog:
        cout << "My pet is a dog";
        break;
    // etc
    }
```

Spørsmål?

`enum` vs. `enum class`

- Hittil har vi egentlig sett på scoped `enum` som har syntaks `enum class`.
- Vi har også det som heter plain enum

```
enum Color {red, green, blue};
enum class Animal {cat, dog, bird};

int main() {
    cout << red << endl;     // 0
    cout << cat << endl;     // feilmelding
}
```

- `enum class` er strengere, og foretrekkes.

enum vs. enum class

```
enum Color {red, green, blue};
enum class Animal {cat, dog, bird};
int main() {
    Color color {red};
    Animal animal {Animal::cat};
    if (color == red) {
        cout << "red";
    }
    if (color == 0) {
        cout << "red";
    }
    if (animal == Animal::cat) {
        cout << "cat";
    }
}
```

- Bruk **enum class**!

Livekoding

Plan for dagen

Motivasjon for øvingen

enum

Klasser

Klasser vs structs

Klasser - intro

- En klasse er en egendefinert type, akkurat som `enum` og `struct`
- En klasse er godt egnet til å uttrykke et konsept som har både **tilstand** og **oppførsel**
- Man kan for eksempel definere konseptet menneske, der hvert menneske har:
  - Egne *tilstander*: som navn, høyde, alder, bosted osv.
  - Mulige *oppførsel*: gå, snakke, spise, programmere, osv.
- Med klasser kan vi lage hva som helst! Det er dette som er *objektorientert programmering*.
- `vector, Rectangle` og Circle er eksempler på klasser vi har jobbet med.

Klasser - intro

- Vi skiller mellom å *definere* en klasse og å *bruke* den.
- Klassedefinisjonen er en slags oppskrift, som vi kan bruke til å lage objekter.
- Et objekt er en instans av en klasse, dvs: en variabel med klassen som datatype.
- Eksempel:
  - Vi er ulike *instanser* av *konseptet* menneske.
  - `vector<int> vec;` `vec` er en instans/objekt av `vector`-klassen.

Klasse syntaks

- Vi deklarerer en klasse ved å skrive:

```
class MyClass {
private:
    int myMemberVariable;
    void myMemberFunction();
public:
    MyClass(int myVariable);
    MyClass();
    int getMyMemberVariable();
    void setMyMemberVariable(int newValue);
    void printMyVariable();
};
```

Oppdeling av filer

- Klassedeklarasjonen skal stå i headerfilen (.h/.hpp), og medlemsfunksjoner defineres i kilde-filen (.cpp).
- Vi lager en header-fil og en kilde-fil per klasse, hvor filene har samme navn som klassen.
- Det er mulig å definere medlemsfunksjonene direkte i klassedeklarasjonen i header-filen.
- Anbefalt å dele det opp i to filer for å øke leselighet.

Klasser: Klassedeklarasjon

- Klassedeklarasjonen skal stå i headerfilen.
- Her deklarerer vi en klasse som heter `Person` og har medlemsvariablene `name` og `age`:

```
// Person.h
class Person {
    private:
        string name; // Medlemsvariabel
        int age; // Medlemsvariabel
    public:
};
```

Klasser: Objekt/instans av en klasse

- Husk! Klassen vår er bare en "oppskrift" som bestemmer hvordan et objekt av klassen skal se ut og oppføre seg.
- Et objekt eller en instans av en klasse er en variabel som har klassen vår som type.
- Men hvordan skal vi instansiere (opprette) et objekt av klassen? Det er her konstruktøren kommer inn.

Klasser: Konstruktør

- Konstruktøren er en spesiell type medlemsfunksjon som kalles på når vi oppretter et objekt/instans av klassen.
- Konstruktøren beskriver hvordan et objekt av klassen opprettes.
- Konstruktøren har...
  - ingen returverdi.
  - samme navn som klassen.
- En klasse kan ha flere ulike konstruktører, som kan brukes til å opprette et objekt av klassen på ulike måter (f.eks. med ulike inputparametere).

Klasser: Konstruktør

- Oppgaven til konstruktøren er å gi medlemsvariablene en startverdi.
- Konstruktøren til `Person`-klassen må altså gi lovlige startverdier til medlemsvariablene `name` og `age`.

```
// Person.h
class Person {
    private:
        string name; // Medlemsvariabel
        int age; // Medlemsvariabel
    public:
        Person(string n, int a); // Konstruktør
};
```

Klasser: Konstruktør

- Implementasjon av en konstruktør i kildefil (.cpp-fil):

```
// Person.cpp
// Denne konstruktøren tar inn to argumenter.
// Disse brukes til å gi medlemsvariablene en verdi.
Person::Person(string n, int a) {
    name = n;
    age = a;
}
```

- Når vi implementerer medlemsfunksjoner i kildefilen må vi bruke resolusjonsoperatoren `::` for å fortelle datamaskinen at denne funksjonen er en del av klassen.

Klasser: Konstruktør

- Konstruktøren kan også ta i bruk en *initialiseringsliste* for å gi medlemsvariablene en startverdi:

```
// Person.cpp
Person::Person(string n, int a) : name{n}, age{a} {
    // Vi må fortsatt ha med en funksjonskropp.
    // Denne kan enten være tom, eller inneholde kode vi vil at
    // konstruktøren skal kjøre i det den oppretter et objekt av klassen.
}
```

Klasser: Instansiering av et objekt

- Nå som vi har en konstruktør kan vi opprette et objekt av klassen (altså en variabel som har klassen som datatype) i `main()`:

```cpp
// Person.cpp
Person::Person(string n, int a) {
    name = n;
    age = a;
}
```

```cpp
// main.cpp
int main() {
    // p1 er et objekt av typen Person.
    Person p1 {"Ayesha", 22};
}
```

Spørsmål?

Klasser: Default-konstruktør

- En default-konstruktør er en konstruktør som ikke tar inn argumenter.
- Nyttig hvis vi vil at det skal være mulig å opprette et default-objekt av klassen.
- Hvis vi ikke definerer en konstruktør for klassen vår, vil det bli laget en default-konstruktør for oss som initialiserer medlemsvariablene våre til tilfeldige verdier.

```
// Person.h
class Person {
    private:
        string name; // Medlemsvariabel
        int age; // Medlemsvariabel
    public:
        Person(string n, int a); // Konstruktør
        Person(); // Default-konstruktør
};
```

Klasser: Default-konstruktør

- Implementasjon av default-konstruktøren i kildefilen:

```
// Person.cpp
Person::Person(){
    name = "Navn Navnesen";
    age = 0;
}
```

- Medlemsvariablene `name` og `age` blir satt til default-verdier som vi har bestemt.

Klasser: Instansiering av et objekt

- Vi kan nå instansiere objekter av klassen på to ulike måter:

```
// main.cpp
int main() {
    Person p1 {"Ayesha", 22};
    Person p2; // Bruker defaultkonstruktøren
}
```

Spørsmål?

Klasser: Dot-operatoren

- Når vi har laget et objekt av klassen, bruker vi dot-operatoren (`.`) til å få tak i medlemsvariablene til objektet eller for å bruke medlemsfunksjoner på objektet.
- Eksempel: `vec.size();`

Klasser: Dot-operatoren

- Når vi har laget et objekt av klassen, bruker vi dot-operatoren (`.`) til å få tak i medlemsvariablene til objektet eller for å bruke medlemsfunksjoner på objektet, *gitt at vi har lov til å aksessere medlemsvariabelen/medlemsfunksjonen*.

```
// main.cpp
int main() {
        Person p1 {"Ayesha", 22};
        Person p2;

        cout << p1.name; // Vil ikke kompilere
}
```

Klasser: `private` og `public`

- I klassedeklarasjonen skriver vi **`private`** og **`public`** for å indikere hvem som skal ha tilgang til de ulike medlemsvariablene og medlemsfunksjonene.

```
// Person.h
class Person {
    private:
        string name; // Medlemsvariabel
        int age; // Medlemsvariabel
    public:
        Person(string n, int a); // Konstruktør
        Person(); // Default-konstruktør
};
```

Klasser: `private` og `public`

- Medlemsvariabler og medlemsfunksjoner som er `private` er kun mulige å få tak i hvis man er inne i klassen.
- Medlemsvariabler og medlemsfunksjoner som er `public` kan brukes også utenfor klassen (f.eks. fra `main()`).
- Vi ønsker å ha så sikker kode som mulig, der minst mulig kan gå galt.
  - Det aller tryggeste er om ingen kan få tak i medlemsvariablene eller medlemsfunksjonene våre - for da kan de ikke gjøre noe galt!
  - Tommelfingerregel: Bare bruk `public` når du må.

Klasser: `private` og `public`

- Medlemsvariabler og medlemsfunksjoner som er **`private`** er kun mulige å få tak i hvis man er inne i klassen.
- Medlemsvariabler og medlemsfunksjoner som er **`public`** kan brukes også utenfor klassen (f.eks. fra `main()`).
- Vi ønsker å ha så sikker kode som mulig, der minst mulig kan gå galt.
  - Det aller tryggeste er om ingen kan få tak i medlemsvariablene eller medlemsfunksjonene våre - for da kan de ikke gjøre noe galt!
  - Tommelfingerregel: Bare bruk **`public`** når du må.
- Men hva om vi vil kunne lese eller oppdatere verdien til en **`private`** medlemsvariabel? Løsning: Lag medlemsfunksjoner!

Klasser: medlemsfunksjoner

- Medlemsfunksjoner er funksjoner som deklareres som en del av klassen, og som kan brukes på instanser/objekter av klassen.
- Med unntak av konstruktøren må alle medlemsfunksjoner ha en returtype, slik dere er vant med fra vanlige funksjoner.
- I en medlemsfunksjon har du tilgang på de andre medlemsvariablene og medlemsfunksjonene i klassen.
- Hvis en medlemsfunksjon skal kunne brukes utenfor klassen (f.eks. i `main()`) må den være `public`. Kan også ha `private` medlemsfunksjoner som kun kan brukes av andre medlemsfunksjoner i klassen.

Klasser: `get()`- og `set()`-funksjoner

- Vanlig å lage egne medlemsfunksjoner for å aksessere private medlemsvariabler.
  - Blir sikrere enn å gi direkte tilgang til medlemsvariablene, siden vi har kontroll over hva som gjøres i funksjonene.
  - Kan legge inn kode som sjekker at medlemsvariabelen ikke settes til en ulovlig verdi.
- Funksjoner som brukes for å lese verdien til en medlemsvariabler kalles ofte get-funksjoner.
- Funksjoner som brukes for å gi medlemsvariabler nye verdier kalles ofte set-funksjoner.
- Både get- og set-funksjoner må være `public` siden vi vil bruke dem utenfor klassen (f.eks. i `main()`).

Klasser: `get()`- og `set()`-funksjoner

- For å gjøre koden mer leselig er det vanlig å navngi get- og set-funksjoner på denne formen: `getVariabelnavn()` og `setVariabelnavn()`.

```
// Person.h
class Person {
    private:
        string name; // Medlemsvariabel
        int age; // Medlemsvariabel
    public:
        Person(string n, int a); // Konstruktør
        Person(); // Default-konstruktør

        string getName(); // get-funksjon for medlemsvariabelen name
        void setAge(int newAge); //set-funksjon for medlemsvariabelen age
};
```

Klasser: `get()`-funksjoner

- En get-funksjon brukes for å kunne lese ut verdien til en medlemsvariabel (her: `name`).
- En get-funksjon returnerer vanligvis en kopi av medlemsvariabelen.

```
// Person.cpp
string Person::getName(){
    return name; // returnerer en kopi av medlemsvariabelen name
}
```

Klasser: `get()`-funksjoner

```cpp
// Person.cpp
string Person::getName() {
    return name;
}
```

```cpp
// main.cpp
int main() {
    Person p1 {"Ayesha", 22};

    // name er private
    // cout << p1.name; // vil ikke kompilere

    cout << p1.getName();
}
```

- Hva blir skrevet til skjerm?

Klasser: `get()`-funksjoner

```cpp
// Person.cpp
string Person::getName() {
    return name;
}
```

```cpp
// main.cpp
int main() {
    Person p1 {"Ayesha", 22};

    // name er private
    // cout << p1.name; // vil ikke kompilere

    cout << p1.getName();
}
```

- Hva blir skrevet til skjerm?
  Ayesha

Spørsmål?

Klasser: `get()`- og `set()`-funksjoner

- HUSK! Det er ingen magi knyttet til funksjonsnavnet på get- og set-funksjoner.
- De to funksjonene under er begge get-funksjoner som kan brukes til å lese verdien til medlemsvariabelen `name`.

```
string Person::getName(){
    return name;
}
```

```
string Person::jegErEnFunksjon() {
    return name;
}
```

- Eneste forskjellen mellom disse funksjonene er at `getName()` har et navn som gjør det lett å skjønne hva funksjonen gjør uten å måtte se på implementasjonen.

Klasser: `set()`-funksjoner

- En set-funksjon brukes for å kunne endre verdien til en medlemsvariabel (her: age) på en lovlig måte.
  - En set-funksjon returnerer som regel ingenting.
- Sikrere enn å bare la medlemsvariabelen endres direkte, for nå kan vi legge inn kode som sjekker at verdien vi vil bruke er lovlig:

```
// Person.cpp
void Person::setAge(int newAge){
    if (newAge >= 0) { // Er den nye verdien lovlig?
        age = newAge; // Oppdaterer verdien til medlemsvariabelen age
    }
}
```

Klasser: `set()`-funksjoner

```cpp
// Person.cpp
void Person::setAge(int newAge) {
    if (newAge >= 0) {
        age = newAge;
    }
}
```

```cpp
// main.cpp
int main() {
    Person p1 {"Ayesha", 22}; // age = 22

    p1.setAge(23); // age = ??
}
```

- Hva er verdien til `p1` sin age-variabel?

Klasser: `set()`-funksjoner

```cpp
// Person.cpp
void Person::setAge(int newAge) {
    if (newAge >= 0) {
        age = newAge;
    }
}
```

```cpp
// main.cpp
int main() {
    Person p1 {"Ayesha", 22}; // age = 22

    p1.setAge(23); // age = 23
}
```

- Hva er verdien til `p1` sin age-variabel?

Klasser: `set()`-funksjoner

```cpp
// Person.cpp
void Person::setAge(int newAge) {
    if (newAge >= 0) {
        age = newAge;
    }
}
```

```cpp
// main.cpp
int main() {
    Person p1 {"Ayesha", 22}; // age = 22

    p1.setAge(23); // age = 23

    p1.setAge(-15); // age = ??
}
```

- Hva er verdien til `p1` sin age-variabel?

Klasser: `set()`-funksjoner

```cpp
// Person.cpp
void Person::setAge(int newAge) {
    if (newAge >= 0) {
        age = newAge;
    }
}
```

```cpp
// main.cpp
int main() {
    Person p1 {"Ayesha", 22}; // age = 22

    p1.setAge(23); // age = 23

    p1.setAge(-15); // age = 23
}
```

- Hva er verdien til `p1` sin age-variabel?

Livekoding

Plan for dagen

Motivasjon for øvingen

enum

Klasser

Klasser vs structs

Klasser vs structs

- Structs og klasser er nesten ekvivalente i C++
  - I klasser er alle medlemsfunksjoner og medlemsvariabler `private` med mindre noe annet er spesifisert.
  - I `structs` er alle medlemsfunksjoner og medlemsvariabler `public` med mindre noe annet er spesifisert.
- I C++ er det vanlig å bruke
  - ... klasser når vi ønsker mer komplekse strukturer, med både medlemsvariabler og medlemsfunksjoner.
  - ... `structs` når vi ønsker å lage en struktur som bare består av data (medlemsvariabler).

Spørsmål?