TDT4102
Prosedyre- og objektorientert programmering

Egendefinerte datatyper:
struct og klasser
og litt mer

Dagens forelesning

- Enumeration typen (kap. 2.2)
  - og litt repetisjon om konstanter
- Egendefinerte datatyper
  - struct og class (strukturer og klasser)

Konstanter (repetisjon)

- Konstanter er ”faste” verdier
  - Spesifikke verdier som vi bruker ofte i koden vår
  - Eksempelvis MAKS og MIN
- Deklareres ved at vi bruker nøkkelordet const
  - Vanlig praksis å bruke store bokstaver for KONSTANTER

```
const double PI = 3.14159;
```

- Fordeler:
  - Enkelt å oppdatere koden
  - Vi kan referere til spesifikke verdier vha. et navn
  - Garanterer at vi bruker samme verdi overalt

enumeration-typer (kap. 2.3)

- Kan kalle det «enum-type»
- En samling av ”symbolske” verdier
  - Symbolsk fordi vi bare trenger en verdi som kan representere noe
  - Eks: Måneder, dager i uka, kompassretninger, farger
- En enumeration-type er en datatype for ett avgrenset sett med konstanter
- Vi kan definere våre egne typer ved hjelp av enum (enumeration)

```
enum  Color {BLACK, WHITE, GREEN, RED, YELLOW, BLUE};
```

enum

type og variabler

- Typen er egentlig bare et sett av heltalls-konstanter som er koblet til et typenavn
- Gir oss en kontrollmekanisme for hva som er lovlig verdier for variabler av denne typen
- Men også nyttig fordi det forenkler programmeringen
  - Det er enklere å skrive ord som symboliserer gitte verdier enn å huske tallverdier, teksttegn eller strenger - og hva de betyr
- En enum-variabel kan kun lagre konstanter som er definert i typen

```
Color c1 = BLUE;
Color c2 = BLACK;
```

Konstantverdiene i enum

- Hvis enum kun er definert med navn blir verdiene satt automatisk (første = 0, andre = 1 osv.)

```
enum  Color {BLACK, GREY, WHITE};
```

- Eller vi kan sette verdier selv

```
enum  Color {BLACK = 0, GREY = 50, WHITE = 100};
```

- alle verdier kan settes eksplisitt
- eller vi kan sette verdi for noen, og utelate andre
- husk at enum egentlig bare er heltall!

Sammensatte datatyper

- Basisdatatyper => enkeltverdier
- Arrays => flere verdier av samme type
- Men mange av de ”data” vi bruker er komplekse:
  - Data som hører logisk sammen, men består av forskjellige typer
- Den enkle løsningen: egendefinerte strukturer
  - struct (og union)
  - del av programmeringsspråket C
- Den mer avanserte løsningen:
  - Klasser
  - Del av programmeringsspråket C++

struct – kort oversikt

- struct -> samling av verdier som kan være av forskjellig type
- (array -> samlinger av verdier av *samme* type)
- Håndteres som en enkelt variabel, med en egen notasjon (.) for å lese/skrive enkeltverdiene
- Vi definerer selv hvilke typer verdier vi har behov for å "samle" i en struktur
- Typen må deklareres før den kan brukes

struct typer

- Deklareres globalt (vanligvis)
- struct er en typedefinisjon
- Typenavnet brukes som vanlig i deklarasjoner av variabler

```
enum Color {BLACK, GREY, WHITE};

struct Circle {
    double radius;
    Color color;
};

int main( ){

    // Enkelt-variabel
    Circle c;

    // Array av Circle
    Circle circles[100];

}
```

struct-variabler

- Kan initialisere med verdier (i riktig rekkefølge)
- Bruke tilordning
- Ha som parameter i funksjoner (som verdi, peker, referanse) – og ha som returtype.

```
Circle biggestOfTwo(Circle a, Circle b);

int main( ){

    Circle c1 = {5.1, WHITE};
    Circle c2 = {8.2, BLACK};
    Circle c3 = biggestOfTwo(c1, c2);

}
```

struct verdiene

- Enkeltverdiene i en struct-variabel kalles medlemmer
  - Egne variabler (som hver er en del av strukturen)
- Medlemmer aksesseres med medlemsoperatoren
  - variabelnavn.medlem

    (vi bruker punktum mellom variabelnavn og medlemsnavn)

```
c1.radius = 5;
c1.color = GREY;
```

- Både "hele" variabelen og medlemsvariablene kan brukes på samme måte som variabler av basis-datatyper
  - i tilordning
  - i kall til funksjoner enten verdi, peker eller referanse
  - som returtype

Men noen begrensinger

- Mange operatorer støtter bare basis-datatypene (int, char, double, bool)
- struct-typer vi lager er ”ukjente” for operatorer som * + - < > == .....
- Seinere skal vi lære hvordan vi kan overlagre slike operatorer til å støtte egendefinerte typer

Litt administrativt

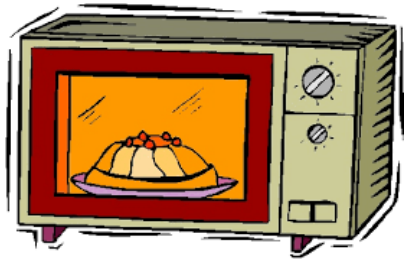
- Læreboka utsolgt!
  - Akademika jobber med saken
  - Status
  - "det ordner seg" ☺
- Referansegruppen
  - En til meldte seg
  - Møte om ca. 2 uker
- Husk "Hjelp i R1"
  - Se "Hva Skjer…?"
  - Neste onsdag:
    - (1) Pekere, (2) Linux og Make
  - Hver onsdag heretter
    - Varierte tema
    - Målsetting 100 – 200 studenter møter opp

Klasser

- Hovedbegrep i objektorientert programmering
- En klasse har typisk både medlemsvariabler og medlemsfunksjoner
- *Klasser er egentlig ganske like struct*
  - *En struct kan også ha funksjoner som medlemmer*
  - *MEN: I en struct er medlemmene default public, mens i en class er medlemmene default private*
  - *Også noen andre forskjeller*
- *For enkelthets skyld er det greit å skille mellom*
  - *Struct som en sammensatt datatype*
  - *Klasse som den objektorienterte konstruksjonen for å samle data og funksjoner*

Objektorientering

- I første omgang ser vi på: klasser og objekter
- En klasse er en datatype
- Et objekt er en verdi av denne type
  - Objekter (variabler) kalles ofte for instanser
- En ny måte å tenke på, men med prosedural programmering i bunnen
- Forenkler programmeringen av data og funksjonalitet vi trenger for disse data'ene
- Meget viktig for abstraksjon!
  - Viktig i de fleste fag (som er komplekse)
  - Krever en viss grad av modning



Klassedeklarasjonen

(vi går tilbake til sirkel-eksempelet)

- En egen (selvdefinert) datatype for sirkler
  - kan ha variabler av sirkeltypen og funksjoner som kan benyttes på disse
- I første omgang kun en verdi for radius og funksjoner som beregner areal og omkrets

Ved hjelp av struct-type og separate funksjoner

```
struct Circle {
  double radius;
};

double area(Circle c);
double circumference(Circle c);
```

Ved hjelp av class kan vi definere data elementene og funksjonene som en samlet type

```
class Circle{
public:
  double radius;
  double area();
  double circumference();
};
```

Bruk av klasser


```
Circle a;
Circle b;
```

gir to objekter av typen Circle; som lagres i variablene "a" og "b"

- Objekter har:
  - Data (medlemmene)
    - radius
  - Operasjoner (medlemsfunksjoner/metoder)
    - area() og circumference()
- Tilgang til medlemsvariabler som for struct
  - ved hjelp av "." a.radius, b.radius

Bruk av klasser

- En klasse er en fullverdig datatype
  - På linje med int, double, char – samt struct-typer du selv definerer
- Kan brukes som parameter
  - Call-by-value
  - Peker, referanse
- Kort sagt: vi kan bruke klasser som en hvilken som helst annen datatype

Medlemsfunksjoner

- Kalles vha. "**.**"-notasjonen:

```
std::cout << a.area();
```

- Implementeres separat fra klassedefinisjonen
  - f.eks. etter main()
  - bruke "**::**" for å koble funksjonens implementasjon til riktig klasse (scope resolution operator)

```
double Circle::area(){
    return PI * radius * radius;
}
```

- Legg merke til at vi kan bruke radius-variabelen i funksjonsimplementasjonen *uten* bruk av parameter
  - funksjonen vet "hvilken" radius-variabel den skal aksessere

public og **private**

- **public** og **private** er modifikatorer som vi bruker for å definere hva som skal være synlig eller ikke
  - for ″ekstern kode″
- Det som er **private** kan bare benyttes av klassens egne funksjoner
- Det som er **public** kan benyttes av kode som er ekstern i forhold til klassen
  - For eksempel kode i main() eller andre funksjoner som IKKE er medlemsfunksjoner

Innkapsling med public og private

- Husk:
  - alt som er **public** i en klasse er "synlig" for annen kode
- I objektorientering ønsker vi innkapsling av data
  - skjule dataelementene for andre enn klassens egne funksjoner/metoder
  - forhindre uvøren/uheldige endringer av objektets data m.m.
- Dette gjør vi ved å spesifisere at medlems*variablene* er **private** mens medlems*funksjonene* er **public**

```
class Circle{
private:
    double radius;
public:
    double area();
    double circumference();
};
```

- Effekten (og hensikten) av dette er at vi ikke lenger kan endre objektenes variabler utenfra (kun via medlemsfunksjonene)

MEN: hvordan kan vi da sette verdier, lese verdier, endre verdier

- I henhold til prinsippet om innkapsling gjøres dette kun med medlemsfunksjoner
- Vi kan lage slike funksjoner for å sette og lese verdier
- Og vi kan kontrollere hva det skal være lov til å gjøre med dataene
  - f.eks. teste på gyldige verdier
- For å sette og lese verdier bruker vi hhv:
  - set-metoder
  - get-metoder

```
class Circle{
private:
   double radius;
public:
   void setRadius(double r);
   double getRadius();
   double area();
   double circumference();
};
```

implementasjonene

```
void Circle::setRadius(double r){
    radius = r;
}
double Circle::getRadius(){
    return radius;
}
```

Objektorientert programmering

- Er å tenke klasser og objekter
  - Dele opp funksjonaliteten i håndterbare medlemsfunksjoner
- Etter hvert skal vi se at samhandling mellom objekter er viktig
- Gjør det enklere å håndtere data samt å kombinere data og funksjonalitet

Klasser i C++ bibliotekene

- Vi finner mange nyttige klasser i C++ bibliotekene
- Vi har allerede benyttet cin og cout mye
  - disse er objekter: instanser av klasser!

    istream cin;

    ostream cout;
  - vi har ikke brukt medlemsfunksjoner, men operatorene << og >>
- En annen nyttig klasse er string-klassen i biblioteket <string>

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

int main(){
    string s1("Hello");
    string s2 = "World";

    string s3 = s1 + " " + s2 + "!";
    string s4 = "Strange";
    s3.insert(6, s4);

    cout << s3 << endl;
    cout << "Lengde = " << s3.size();
}
```

string-klassen
C++ standard klasse

- Definert i biblioteket <string>
- string-objekter håndteres delvis likt med basis typene
- Tilordning, sammenligning, konkatenering

```
string s1, s2, s3;
s3 = s1 + s2;
s3 = "Hello World!";
cout << s3 << endl;
```

Små eksempel

Noen kodebiter lagt ut på its learning

Gjennomgått raskt

NTNU

STØRRE GJENNOMGÅENDE EKSEMPEL: «LASSE'S ISKREMKIOSK»

Eksempel

- «Discrete Event Simulation»
  - En programvareteknikk
  - Er kjernen (motor) i de fleste dataspill
  - Sentral i svært mange ingeniør-anvendelser
    - Simulering elektriske kretser
    - Vindmølle-parker
    - SmartGrid
    - …

- Klasse: Hendelse (Event)
  - Student vil kjøpe is og stiller seg i kø ved tid t1
  - Kiosken lager og selger deg en is ved tid t2
  - Hver hendelse et objekt (klasse-instans)
- Simulering: behandle kø av hendelser som er sortert på tid

Eksempel

Visual Studio

Viste påbegynt eksempel, som brukte noen av begrepene vi har lært denne uken

Kommer nye og bedre versjoner av dette etter hvert ☺