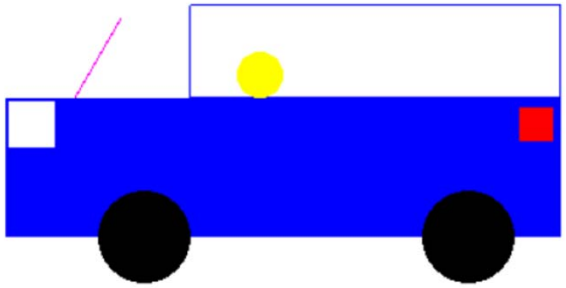
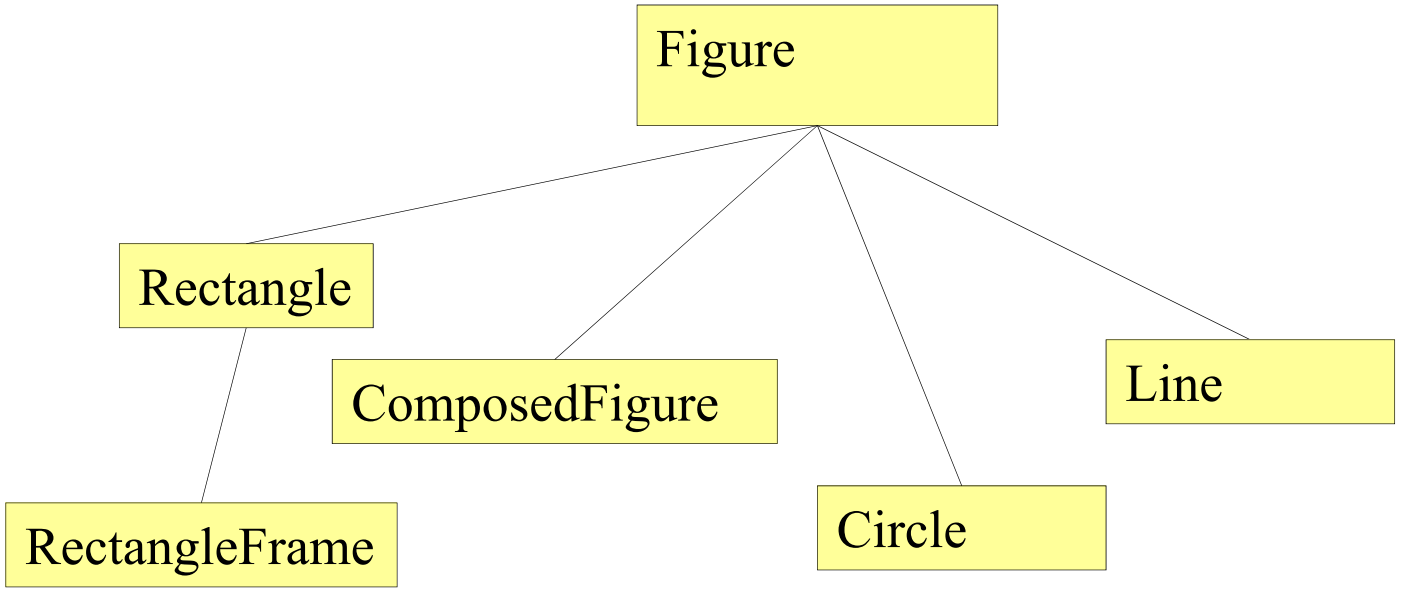
TDT4102
Prosedyre- og objektorientert programmering, Uke 11

Arv,
Virtuelle funksjoner og polymorfi,
Grafikk, eksempel (SFML)





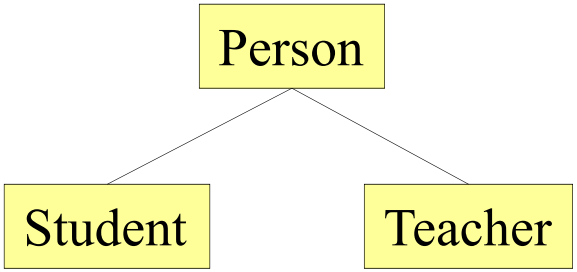
Ukas forelesning

Kap. 14, 15

- Arv
  - Repetisjon
  - Konstruktører og tilordningsoperator, eksempel
- Nytt stoff
  - virtuelle funksjoner / dynamisk binding
  - abstrakte klasser
- Eksempel
  - «Større» eksempel inspirert av læreboka (Grafikk m/SFML)
- Kahoot (enklere enn før)
- C++11 og eksamen

Arv i programmering

- Lage nye klasser som er utvidelser av andre klasser
  - ① Bygge på eksisterende klasser ved å legge til nye medlemsvariabler og medlemsfunksjoner
  - ② Spesialisere eksisterende funksjoner ved å **redefinere** (lage nye implementasjoner av) eksisterende funksjoner



Eksempel fra forrige time (KlasserArv.cpp)

```
class Person{
private:
    string name; //name
    string ssn;  //social security number

public:
    string getName() const;
    void setName(const string &name);
    string getSsn() const;
    void setSsn(const string &ssn);
    string toString() const;
};
```

```
class Student: public Person{
private:
    string sid;     //student id
    string program; //study program
public:
    string getSid() const;
    void setSid(const string &sid);
    string getProgram() const;
    string setProgram(const string &program);
    string toString() const;
};
```

Redefinering av toString() i subklasse

```
string Student::toString() const {
    return "Student: " + Person::toString()
                + ", sid = " + sid;
}
```

Bruker her scope-operatoren KlasseNavn:: for å kalle superklassens toString - funksjon

Konstruktører

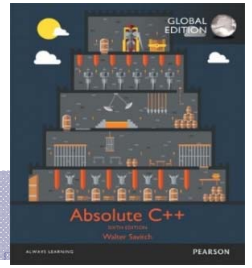
- Konstruktører (arves ikke)
  - Default: kall til superklassens default-konstruktør
  - Gjør helst: Eksplisitt kall til riktig konstruktør fra superklassen i initialiseringslista til subklassen

```
Person::Person(string name, string ssn)
        : name(name), ssn(ssn) {
}
```

```
Student::Student(string name, string ssn, string sid):
        Person(name, ssn), sid(sid) {
}
```

- Kopikonstruktør (arves ikke)
  - På samme måte (automatisk kall til superklassens default konstruktør, med mindre eksplisitt kall til spesifikk konstruktør i initialiseringslista)

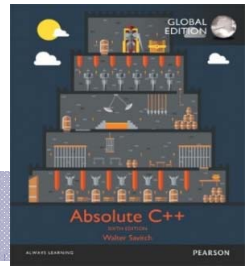
Kap. 14.2



Tilordningsoperator

- Tilordningsoperator arves heller ikke
  - Genereres en default hvis du ikke implementerer en egen
  - Lag din egen tilordningsoperator der du også bruker superklassens tilordningsoperator

Kap. 14.2




```
Student& operator=(const Student& rhs){
    Person::operator=(rhs);
    // … utfører tilordningsoperator for den avledete klassen (Student)
    return *this;
}
```

Vi bruker superklassens tilordningsoperator som sørger for at superklassens medlemmer blir oppdatert (for this object) med verdier fra rhs.

Eksempel --- KallsekvensV2.cpp

```
class Person {
private:
    string name;
public:
    Person(string name) : name(name) {
        cout << "Person - constructor" << endl;
    }
    Person(const Person& other) {
        name = other.name;
        cout << "Person - copy constructor" << endl;
    }
    ~Person() {
        cout << "Person - destructor" << endl;
    }
    Person& operator=(const Person& rhs) {
        this->name = rhs.name;
        cout << "Person - assignment-operator" << endl;
        return *this;
    }
```

KallsekvensV2.cpp, subklasse student

```
class Student : public Person {
private:
    int id;
public:
    Student(string name, int id) : Person(name), id(id) {
        cout << "Student - constructor" << endl;
    }

    Student(const Student& other) : Person(other) {
        id = other.id;
        cout << "Student - copy constructor" << endl;
    }

    ~Student() {
        cout << "Student - destructor" << endl;
    }

    Student& operator=(const Student& rhs) {
        Person::operator=(rhs);
        this->id = rhs.id;
        cout << "Student - assignment-operator" << endl;
        return *this;
    }
```

KallsekvensV2.cpp, kjøring

```
int main() {
    setlocale(LC_ALL, "Norwegian");
    cout << endl << "Oppretter objekt a og c" << endl;
    Student a("a", 100);
    Student c("c", 300);
    cout << "a: " + a.toString() + "c: " + c.toString();

    cout << endl << "Oppretter objekt b med kopikonstruktør" << endl;
    Student b(a);
    cout << "b: " + b.toString();

    cout << endl << "Tilordner d = c" << endl;
    Student d("d", 400);
    d = c;
    cout << "d: " + d.toString();

    cout << endl << "Går ut av scope" << endl;
}
```

Autos

Name	Value	Type
a	{id=100 }	Student
Person	{name="a" }	Person
id	100	int
b	{id=100 }	Student
Person	{name="a" }	Person
id	100	int

```
Oppretter objekt a og c
Person - constructor
Student - constructor
Person - constructor
Student - constructor
a: student a 100
c: student c 300

Oppretter objekt b med kopikonstruktør
Person - copy constructor
Student - copy constructor
b: student a 100

Tilordner d = c
Person - constructor
Student - constructor
Person - assignment-operator
Student - assignment-operator
d: student c 300

Går ut av scope
Student - destructor
Person - destructor
```

Dagens terminologi
(Oversikt)

- **Polymorfi**
  - Noe som har forskjellige former
- **Virtuelle funksjoner**
  - Funksjoner hvor det bestemmes ved kjøretid hvilken ”versjon” av en funksjon som skal utføres (gjelder redefinerte funksjoner) (kalles også sen binding eller dynamisk binding)
- **Abstrakte klasser**
  - Superklasse hvor minst en funksjon er deklarert, men ikke implementert («A pure virtual function»)
  - Er kun ment som en datatype
  - Klasse som det ikke kan lages instanser av

Virtuelle funksjoner

- Nøkkelordet **virtual** foran funksjons-deklarasjon endrer mekanismen bak valg av funksjonsimplementasjon ved kjøretid
- Bestemmer ved kjøretid hvilken funksjon som skal kalles
  - kalles ”late/dynamic binding”
  - programmet vil finne redefinerte funksjoner selv om vi gjør kall via superklasse-pekere/referanser

Abstrakte klasser

- Funksjoner kan også være deklarert som ”helt” virtuelle («pure virtual») --> har ingen implementasjon

```
virtual string toString() = 0;
```

- Klasser med en eller flere slike funksjoner kalles abstrakte klasser fordi de er ”ufullstendige”
  - kan ikke instansieres fra (dvs. ikke lage objekter)
  - kan likevel brukes som ”type” og vi kan lage implementasjoner som benytter den virtuelle funksjonen

```
string Person::printMe(){
   return "This person " + toString();
}
```

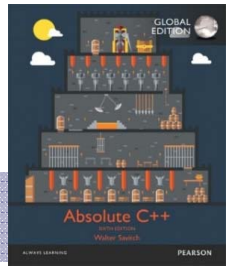
Virtuelle funksjoner

- En gang virtuell -> alltid virtuell
  - Den virtuelle egenskapen arves av alle subklasser
- Men god praksis å inkludere nøkkelordet i alle subklasser for å øke lesbarhet (dokumentere)
- Virtuelle funksjoner fungerer kun sammen med pekervariabler og referanser!

```
vector<Person*> polypersons; //støtter polymorfi
vector<Person> copypersons; //her er alle konvertert til Person
```

Virtual destruktor

Side 716-717

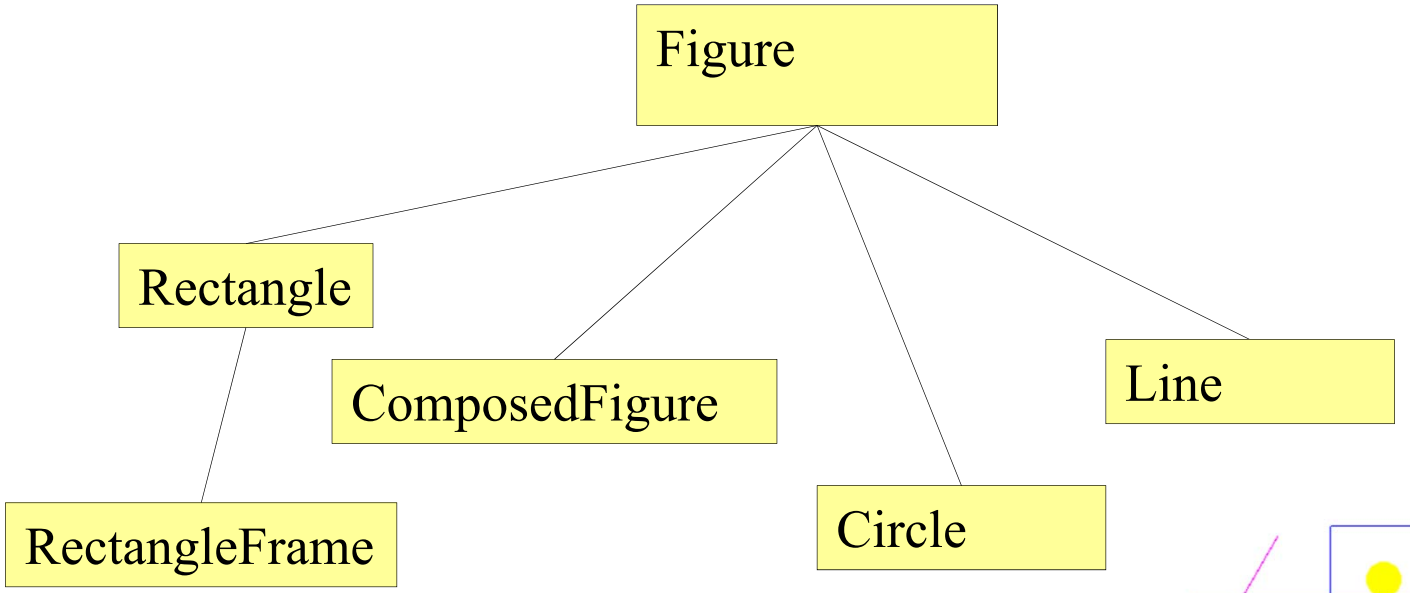


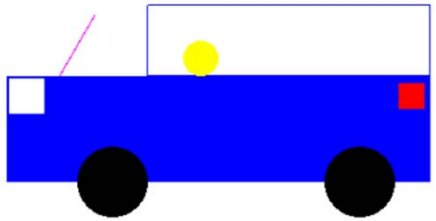
- Klasser som brukes som superklasse i et hierarki bør vanligvis ha destruktor deklarert som virtuell
- For å sikre at riktig destruktor blir kalt (uavhengig av hvilken peker/referanse som er brukt)
- Husk at superklassens destruktor blir kalt implisitt

upcasting/downcasting

- **upcasting** er å tilordne en subtype til en variabel av en supertype
  - helt OK å gjøre dette
  - pekervariabler/referanser -> gjør at instansens faktiske implementasjon av en funksjon kan anvendes pga. at programmet finner frem riktig implementasjon ved kjøretid
- downcasting er å tilordne en supertype til en variabel av en subtype
  - kan gjøres med dynamic_cast<subtype*>(variabel)
  - men da må subtype stemme og virtuelle funksjoner være brukt
  - ved typefeil vil resultatet bli nullpeker
  - Generelt IKKE anbefalt, men bør vite om det (Læreboka side 718)

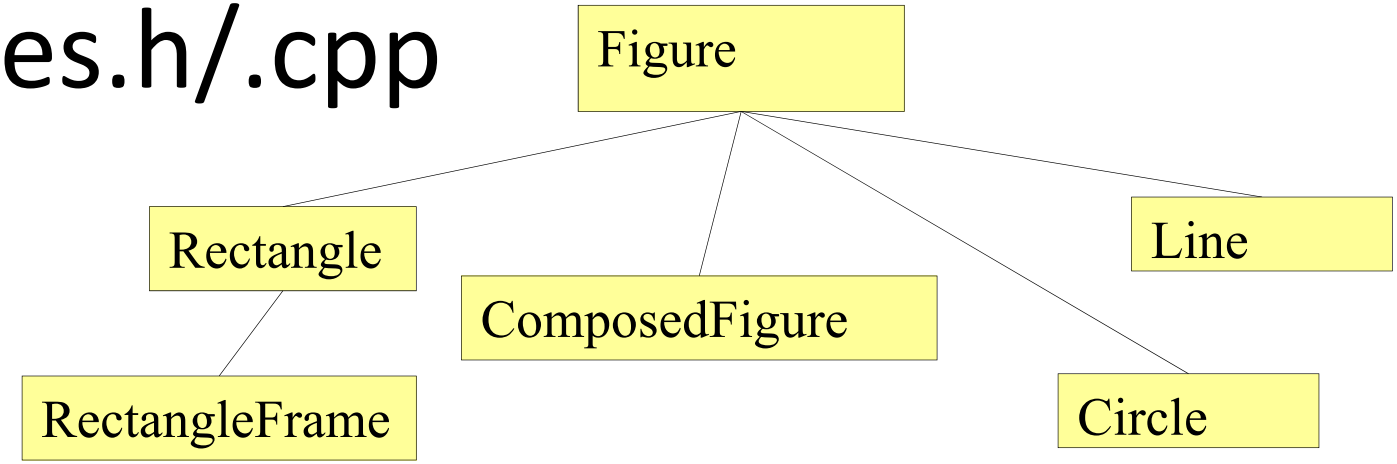
Eksempel: Enkel figurtegning





- Inspirert av læreboka side 698-699 og 708
- Lager et svært enkelt og begrenset bibliotek for figurtegning oppå et mye større og bedre bibliotek (SFML) for det samme
  - Litt kunstig/bakvendt fremgangsmåte
  - Pedagogisk motivert ➔ **konsentrer oss i forelesningen om Figures.h og Figures.cpp**

Figures.h/.cpp



- **Figure**
  - Name, color (protected, default white), pure virtual function draw. Dvs. ikke implementert draw, derfor en abstrakt klasse
- **Rectangle,** «Figure sine data» + Heigth, Width, CenterPoint(x,y). Virtual function draw.
- **Circle ,** Figure + Radius, CenterPoint. vanlig medlemsfunksjon draw.
- **Line,** Figure + Length, StartPoint, Direction. vanlig medlemsfunksjon draw.
- **RectangleFrame,** vanlig medlemsfunksjon draw.

Figures.h/Figures.cpp og SFML

- Klasse Figure
  - Kan bestå av en åpen el. fylt rektangel, en sirkel, en linje, eller sammensetting av disse (ComposedFigure)
- SFML har ikke noe funksjonskall for å tegne en linje
  - Lager primitiv (funksjonskall) **drawLine** (i MySFML.h/.cpp) vha. sf::RectangleShape
- RectangleFrame bruker drawLine
- ComposedFigure
  - Utnytter virtuelle funksjoner (Hovedhensikt med hele eksemplet)

ComposedFigure

```
class Figure {
protected:
    string name; // can be accessed in subclass
    sf::Color color; // color for the figure
public:
    Figure(const string & name, sf::Color color = sf::Color::White);
    virtual ~Figure() {}
    string getName();
    virtual void draw(sf::RenderWindow* win) = 0; // Figure is abstract class
};
```

```
class ComposedFigure : public Figure {
protected:
    vector <Figure*> parts; // vector of Figure components
public:
    ComposedFigure(const string & name);
    ~ComposedFigure();
    void addFigure(Figure *fig);
    virtual void draw(sf::RenderWindow* win);
};
```

ComposedFigure, forts.

```
class ComposedFigure : public Figure {
protected:
    vector <Figure*> parts; // vector of Figure components
public:
    ComposedFigure(const string & name);
    ~ComposedFigure();
    void addFigure(Figure *fig);
    virtual void draw(sf::RenderWindow* win);
};
```

```
void ComposedFigure::addFigure(Figure *fig) {
    parts.push_back(fig);
}
```

```
void ComposedFigure::draw(sf::RenderWindow* win) {
    for (unsigned int i = 0; i < parts.size(); i++) {
        parts[i]->draw(win);
    }
}
```

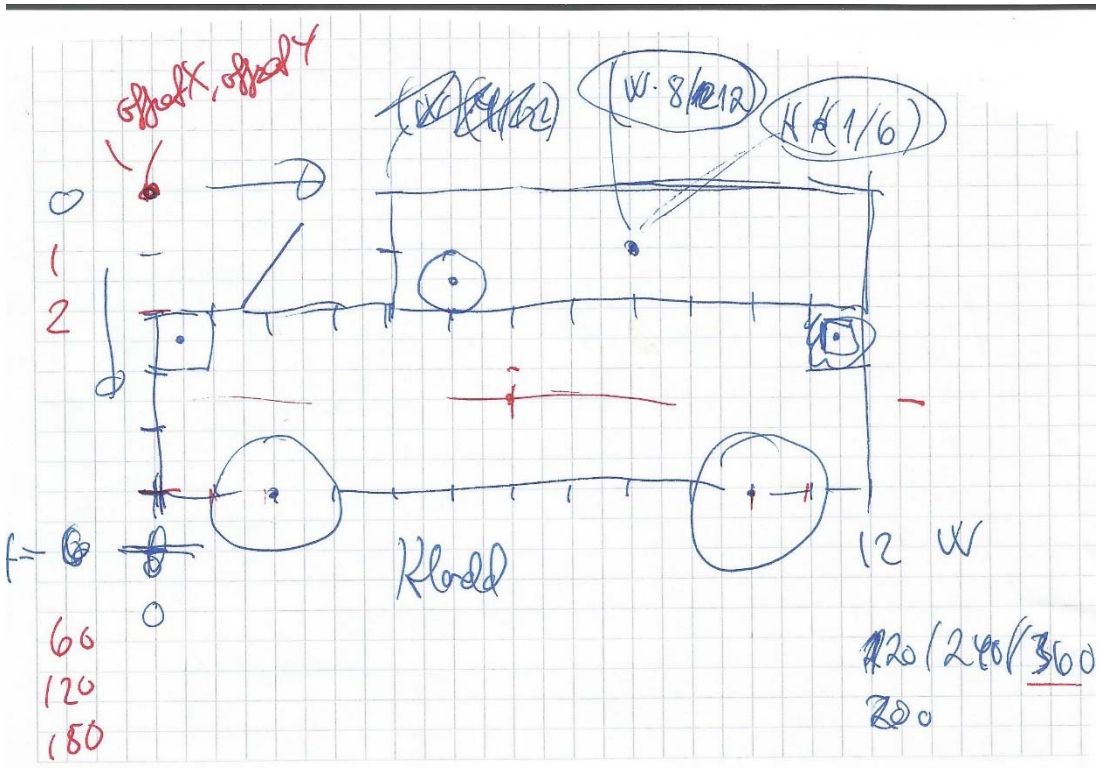
```
ComposedFigure::~ComposedFigure() {
    for (unsigned int i = 0; i < parts.size(); i++) {
        delete parts[i];
    }
}
```

Vis eksempel

* del 1 tester funksjoner,

* del 2 tegner en ganske stygg men «skalerbar» bil

* del 3 viser abstrakt klasse og bruk av virtuell funksjon



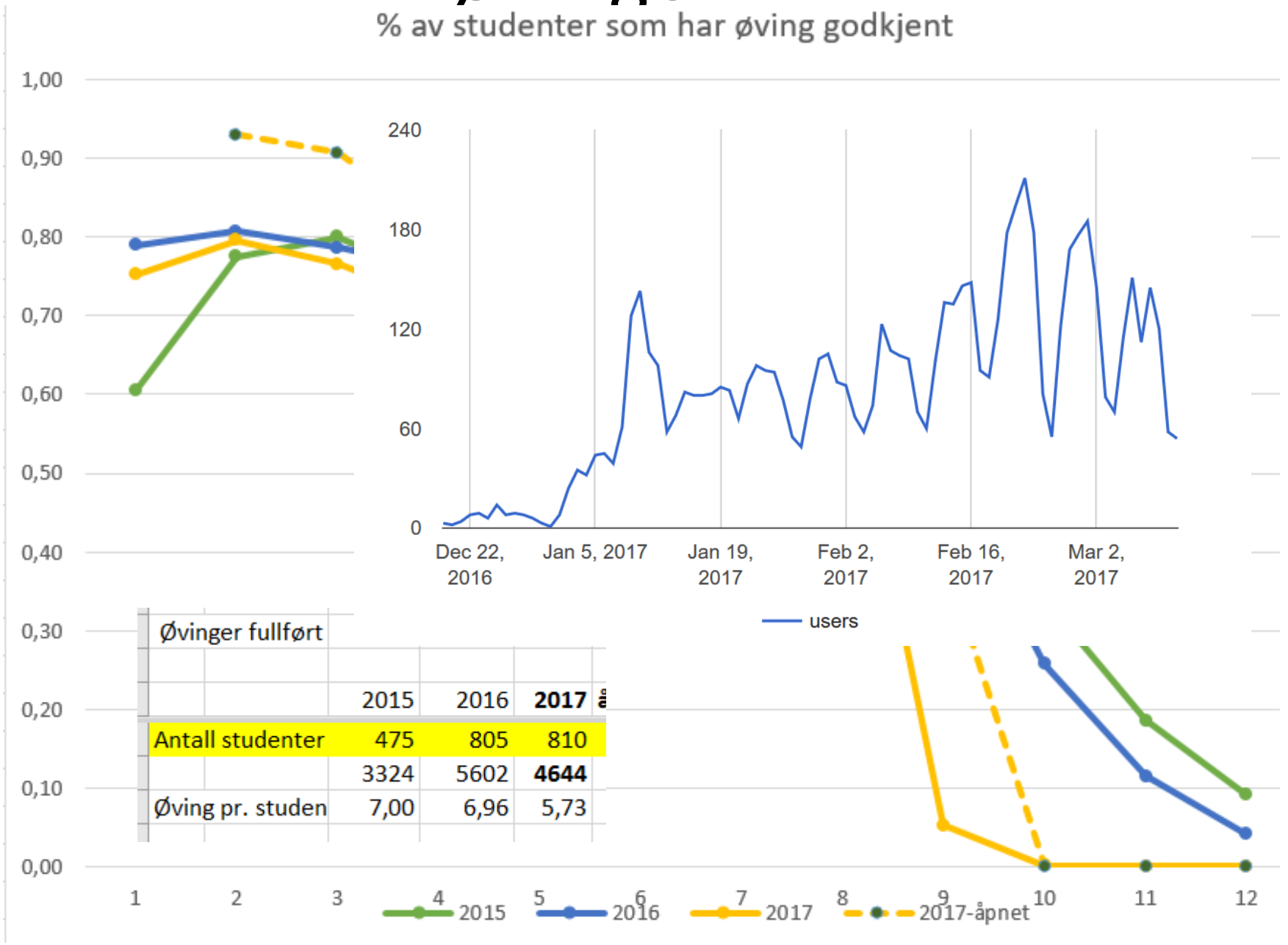
Administrativt

- Referansegruppe
  - Siste 10 minutter i dag
  - Møte 20/3
- Gjesteforelesning 20/3
  - Stian fra ARM
- Spørreundersøkelse om Kahoot, sembly.no, bruk av debugger (kommer)

- CMB-konkurransen (frivillig, ikke pensum)
  - Ca. 15 deltakere, avsluttes i kveld
  - Følg med i CMB-folder under Its'learning for de det gjelder

- Status øvinger

Øvinger



Øvinger fullført

	2015	2016	**2017**
Antall studenter	475	805	810
	3324	5602	**4644**
Øving pr. studen	7,00	6,96	5,73

Repetisjon --- kahoot

«C++ 11 og eksamen»

- Allerede forelest ;
  - Heltalls datatyper <cstdint>
  - **Auto**
  - **Range-based for-loop** med og uten auto
  - «raw string literals»
  - Strong enum (**enum class**)
  - **Medlems-initialisering**
  - **delegerende konstruktør**
  - **Initialiseringsliste**, nullptr
- Appendix om C++11 i 6. utgave av boka, samt litt «her og der»
- Ett ark som vedlegges eksamen, kommer på Its's learning

Initializer lists

```
/* Intialize vec using initializer list */
vector<string> vec = { "AB", "CD", "EF" };
```

auto

```
auto it = vec.begin();
const auto& first = vec[0];
```

range-based for-loops

```
for (string str : vec) {
  cout << str << endl;
}
/* Avoid copy */
for (const string& str : vec) {
  cout << str << endl;
}
/* Avoid copy and use auto */
for (const auto& str : vec) {
  cout << str << endl;
}
```

Enum class (also called strong enum or scoped enumerations)

```
enum class Color : int {red, green, blue};
enum class Feeling : int {sad, happy, blue};
/* Need explicit cast to convert to int */
int x = static_cast<int>(Color::blue);
/* Feelings and Colors can't be compared */
if (Color::blue == Feeling::blue) {
  /* compile error */
}
```

nullptr

```
/* nullptr indicates null pointer value */
int *ptr = nullptr;
```

to_string

```
to_string(23.45); /* -> "23.450000" */
to_string(3) /* -> "3" */
```

Delegating constructors

```
class Student {
  string username;
  string email;
public:
  Student(string username, string email) {
    this->username = username;
    this->email = email;
  }
  Student(string username)
   : Student(username,
             username + "@stud.ntnu.no")
  {}
};
```

Default member initialization

```
class Person {
  bool alive = true;
  /* ... more code ... */
};
```

Unique pointer

`std::unique_ptr` is a smart pointer that owns and manages another object through a pointer and disposes of that object when the `unique_ptr` goes out of scope. In addition, when using C++14 `std::make_unique`, there is no need to manually allocate memory for the object. The `unique_ptr` **owns** the object - it does not allow the pointer to be copied, but the ownership may be transferred using `std::move`. The `unique_ptr` can be used just like a normal pointer, using the indirection operator (`*`) or the arrow operator (`->`).

```
/* create unique_ptr using make_unique - the preferred way */
unique_ptr<Student> s1 = make_unique<Student>("daso");
/* create unique ptr using unique_ptr's constructor */
std::unique_ptr<Student> s2(new Student("lana"));
/* transfer ownership of unique ptr */
auto s3 = move(s1); /* value of s1 is now unspecified */
```

In cases where we want a function to use the object pointed to **without** transferring ownership to the function, we can get the underlying raw pointer

Foreløpig utgave

10 min til referansegruppen