

TDT4102

Prosedyre- og objektorientert programmering

To program is to understand!

Kristen Nygaard (1926–2002)

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

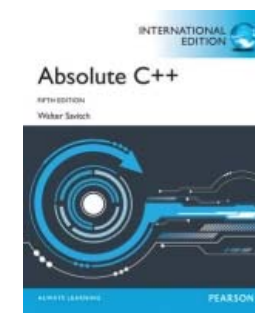
C++

Overlagring av operatorer



Dagens forelesning

- Repetisjon
 - Vector
 - Fra standard template library (STL)
- Ny «teori»
 - Overlagring av operatorer (kap. 8)
 - Hva og hvorfor
 - Syntaks, implementasjon og bruk
- Gjennomgående eksempel, simulering
 - Litt motivasjon – anvendelser
 - Litt bakgrunn for «Discrete Event Simulation (DES)»
 - **Gjennomgang av kode i Visual Studio**
 - **Fokus på å illustrere C++ programmering**



Forelesninger, status, tempo, plan

Fra forelesningsplan på ITSL
(Skulle være 2 uker med ½ fart)

Uke 06	Start operator-overlagring, programmering med klasser	Kap 7. kap. 8 + rep.	Øving 4
Uke 07	Mer operator-overlagring, programmering med klasser	Kap 8	Øving 5
Uke 08	Pekere og dynamiske variabler	Kap 10	Øving 6
Uke 09	I/O streams, filhåndtering, lesing og skriving til/fra fil, rekursjon	Kap 12, 13	Øving 7

Parameteriserte typer (templates)

- Vanlige C++ arrays er en lavnivå konstruksjon som du må kunne, men
- I praktisk programmering bruker vi ofte array, vector og gjerne mer avanserte datatyper som queue, stack, priority queue m.m.m.
 - **Standard Template Library (STL)** (kap. 19)
 - Se f.eks. wikipedia.org/wiki/Standard_Template_Library
- Forrige time, `<vector>` og `<array>` introdusert

Vector

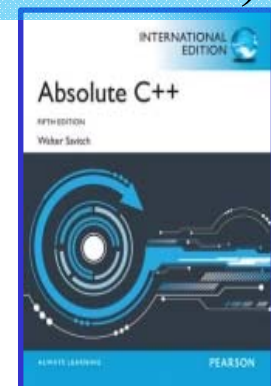
```
#include <iostream>
using namespace std;
int main()
{
    cout << "hello World!\n";
    return 0;
}
```

- Tabeller som kan endre størrelse ved behov
 - Dvs. er fleksibel el. dynamisk
 - Er en klasse, del av standard template library (STL)
 - template-klasser er en type klasser som "tilpasser" seg datatypen de benyttes for
- Legger til elementer i vector med `push_back(type)`
- Lese/skrive verdier med `indeksoperatoren`

```
#include <vector>
```

```
vector<int> myVector = {1, 2, 3};
myVector.push_back(4);
for (int i = 0; i < myVector.size(); i++){
    cout << myVector[i] << endl;
}
```

Kap. 7.3
(Kap. 19 – senere)



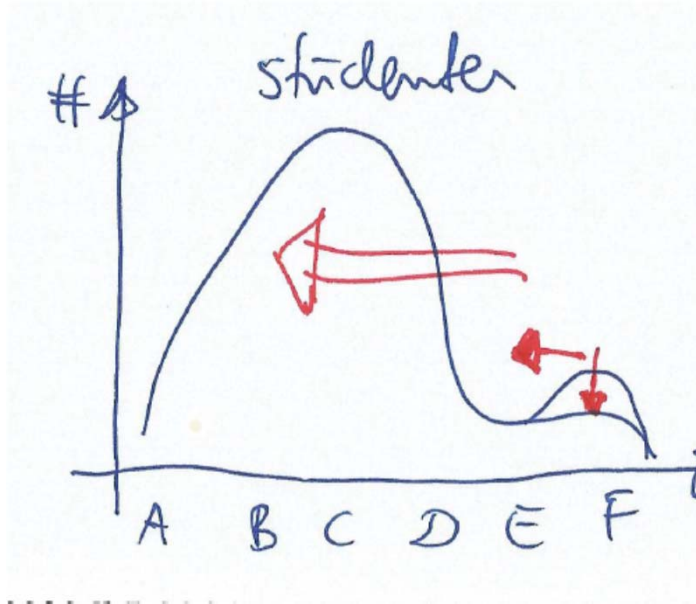
Vector – lett eksempel

- `<vector>`
 - Deklarere
 - Intialisere
 - Utvide
 - Fjerne element
 - Skrive ut
 - Sortere
- Visual Studio
 - Basic_vector.cpp

Operator overloading motivasjon

- Eksempel
 - Vi så at sortering av tall er lett
 - `<vector>`, `sort`
- `<vector>` kan lagre instanser av strukturer og klasser
 - dvs. vilkårlig objekter (av en og samme type)
- Sortering av vector av objekter
 - Veldig ofte nyttig!
 - **Lett hvis vi kan operator-overloading**
- **Først teori, så eksempel**
 - `Sort_int_event.cpp`

Administrativt

- Status lærebok
 - 6. utgave i bokhandel nå (spesialavtale mhp. pris). OK på eksamen
- Ref.gruppe-møte
 - Nyttig, litt sprikende, prøver å tilpasse oss
 - Små vs. større eksempel
- Stor og spredt studentmasse
 - Tilbud til alle
 - Ulike tilbud
 - Hjelp i R1 m.m.
- Ulike læringseksperiment (Se ITSL)
 - «Climbing Mont Blanc»
 - «C++ autograding»
 - ThxBro (On-demand studass) (demo i pausen)

Vikar i neste uke (Johannes), fordi

The screenshot shows a web browser window with the URL `https://agenda.infn.it/contributionDisplay.py?contribId=`. The page title is "Computing on Low-Power Architectures (COLA)". Below the title, it says "25-26 February 2016 IUSS - Ferrara 1391" and "Europe/Rome timezone". On the left, there is a sidebar with links: Overview, Timetable, Contribution List, Travel Information, and Accommodation. The main content area displays a contribution titled "Climbing Mont Blanc - a Prototype System for Training in Energy Efficient Programming". This title is circled in red. Below the title, it says "Presented by Lasse NATVIG on 25 Feb 2016 from 15:00 to 15:25". There is also a "Description" section that starts with "Climbing Mont Blanc (CMB) is an open online judge used for training in energy". On the right, there is a "Place" section with "Location: IUSS - Ferrara 1391", "Address: Via Scienze 41b", and "Room: 1". At the top right of the main content area, there are links for "PDF | XML | iCal".

2D vector

- Eksempel 2D_vector.cpp
 - Ligger i mappen for uke 5

```
mat.size() = 2  
mat[0].size() = 3  
mat[1].size() = 4
```

```
C:\Users\lasse\AppData\Local\Temp\2D_vector.cpp - Notepad++  
File Edit Search View Encoding Language Settings Macro Run Plugins Window ?  
utils.h Source.cpp small input.txt small answer.txt enum.cpp main_new.cpp check  
1 #include <iostream>  
2 #include <vector>  
3 using namespace std;  
4  
5 int main() {  
6     // demo av 2D vector, dvs vector av vector-objekter  
7     vector<vector<int>> mat;  
8     mat.push_back(vector<int>(3));  
9     mat.push_back(vector<int>(4));  
10    cout << "mat.size() = " << mat.size() << endl;  
11    cout << "mat[0].size() = " << mat[0].size() << endl;  
12    cout << "mat[1].size() = " << mat[1].size() << endl << endl;  
13  
C++ source length : 957 lines : 30 Ln : 25 Col : 56 Sel : 0 | 0 Dos\Windows ANSI INS
```

... så, **Ny teori**: Operatorer og overlagring

- Operatorer er en del av programmeringsspråket
 - `+ / * % == < > []` ...med flere
- I praksis kan vi se på operatorer som funksjoner
 - Med en syntaks som vi er kjent med fra andre "språk"
 - og som er integrert del av programmeringsspråket
- `2 + 3` kunne like gjerne vært skrevet som `+(2, 3)`
 - hvor `+` er funksjonsnavn
 - og operandene `2` og `3` er argumenter
 - returnerer summen av beregningen

Overlagring

- Hva er overlagring?
- Vi kan ha flere funksjoner som har samme navn, så lenge parameterlista er forskjellig
- Kompilatoren bruker navn + parameterliste når den skal finne riktig funksjon

På engelsk heter det overloading, men på norsk sier vi enten overlagring eller overlasting

Overlagring av operatorer

- De innebygde operatorimplementasjonene
 - kan i utgangspunktet bare brukes på basis-datatyper
 - kan ikke bruke innebygde operatorer på egendefinerte datatyper – fordi kompilator ikke ”kjenner” dine egendefinerte typer
- I C++ har vi mulighet til å **overlagre operatorer**
 - Lage egne implementasjoner for egne datatyper
 - Kan brukes på egendefinerte typer som klasser (class) og strukturer (struct)
- **Fordel: enkel og intuitiv syntaks, ryddig kode**

Hvorfor?

- Gitt en egendefinert datatype

```
struct Date{
    int year;
    int month;
    int day;
};
```

Vi starter med struct-typer fordi det er enklere å forstå, men vi bruker det oftere for klasser!

- Så hadde det vært greit å kunne skrive

```
Date x = {1982, 10, 11};
Date y = {1984, 11, 12};

if (x == y){
    cout << "x er samme som y" << endl;
}
```

- Sammenligne om datoene er de samme vha. ==
 - I stedet for å måtte skrive en mer detaljert test av enkeltmedlemmene hver gang vi skal sammenligne datoer

Overlagring av operatører

- Lik overlagring av funksjoner
 - Nøkkelordet `operator` og symbolet for operatoren er "navnet" til operatorfunksjonen
 - Har bestandig returtype (kan ikke være void)
 - Antallet operander må være som for eksisterende operator

```
struct Date{  
    int year;  
    int month;  
    int day;  
};
```

(x == y)

```
bool operator == (Date d1, Date d2){  
    return ((d1.year == d2.year)  
        && (d1.month == d2.month) && (d1.day == d2.day));  
}
```

Her tester vi på "int == int"
noe vi allerede har støtte for i C++

Kort forklaring på implementasjonen

- Vi har overlagret operatoren **==** for datatypen struct Date
- I praksis ikke forskjellig fra hvordan vi ville ha implementert en ordinær funksjon for sammenligning
- Den største forskjellen er
 - syntaks for operator deklarasjonen
 - at vi kan bruke standard binær operatorsyntaks for kall

Alle operatører har operander og returtype



- Alle uttrykk i C++ har returverdi
- For mange operatører er det returverdien som er av interesse
 - $(a + b)$ returnerer summen og $(a == b)$ returnerer boolsk verdi
 - men hverken a eller b endres av uttrykket!
- For andre operatører er vi mer interessert i **effekten** operatoren har på operanden
 - $++a$ eller $a += 1$
 - implisitt endring av operanden a
 - men skal også returnere noe
 - f.eks. skal $(++a)$ returnerer verdien etter inkrementering

Implementering av operatorer

- En operator utfører vanligvis en ganske enkel oppgave
- Overlagring av operatorer for en gitt egendefinert datatype består i å:
 - Bestemme hvilke operatorer vi vil overlagre
 - Operandene er i de fleste tilfeller gitt
 - Returtypen er i de fleste tilfeller gitt
- Utfordringen er å lage fornuftig logikk
 - Og riktig bruk av referanseparameter, returtype og const



Bruk av call-by-reference (&)

- Bruker ofte **call-by-reference** i operatorens parameterdeklarasjon
- Noen ganger bare fordi det er litt mer effektivt enn call-by-value

```
bool operator==(Date &d1, Date &d2);
```

 - I dette tilfellet bruker vi call-by-reference kun for å effektivisere
- Andre ganger fordi vi MÅ

```
Date operator++(Date &d);
```

 - operatoren skal endre på operanden
 - Her er det en unær inkrementeringsoperator vi overlager
 - Denne skal både endre operanden og returnere en verdi

Her er det operatoren for prefiks inkrementering vi har overlagret

Bruk av `const` i parameterlista

- Når vi bruker call-by-reference for operandene **kun fordi** vi ønsker effektivitet
- Ønsker vi likevel å sikre oss at implementasjonen ikke endrer på operanden(e)
 - Samt dokumentere for brukere at operandene ikke endres
- Parameterne til operatoren deklarerer da som konstanter
 - Når vi ikke `bool operator==(const Date &d1, const Date &d2);` ønsker **sideeffekter**
 - Sikrer oss at verdiene ikke endres, og øker lesbarheten i koden.

NB! Implementering av sammenligningsoperatorene kan baseres på gjenbruk

```
bool operator< (const Date &d1, const Date &d2){
    if (d1.year < d2.year) {
        return true;
    } else if (d1.year == d2.year) {
        if (d1.month < d2.month) {
            return true;
        } else if (d1.month == d2.month) {
            return d1.day < d2.day;
        } else {
            return false;
        }
    } else {
        return false;
    }
}
```

Vanlig å implementere operatoren < (mindre enn) for deretter å bruke denne i implementasjonen av andre sammenlignings-operatorer

```
bool operator> (const Date &d1, const Date &d2){
    return d2 < d1;
}
```

```
bool operator== (const Date &d1, const Date &d2){
    return !(d2 < d1) && !(d1 < d2);
}
```

```
bool operator<=(const Date &d1, const Date &d2){
    return ((d1 < d2) || (d1 == d2));
}
bool operator>=(const Date &d1, const Date &d2){
    return ((d1 > d2) || (d1 == d2));
}
```

Operator overloading

- motivasjon

- Eksempel
 - Sortering av tall er lett
 - `<vector>`, `sort`
 - Sortering av vilkårlig datastrukturer eller objekter
 - Veldig ofte nyttig!
 - Lett hvis vi kan operator-overloading
- Visual Studio
 - `Sort_int_event.cpp`

Vector vil bli vist også i simulerings-eksempel:

- Dynamisk tabell med hendelser: **vector** **<Event>**
- **Event**, konstruktør, push_back(), begin(), end(), front(), erase() m.m.
- Men først bittelitt bakgrunn om simulering ...

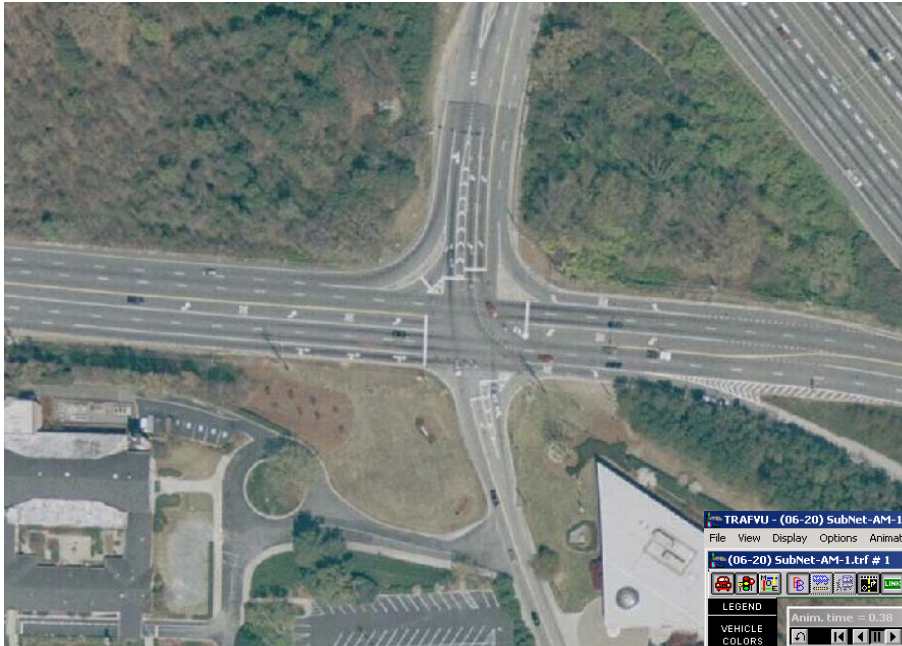
Simulering - motivasjon

*Ikke pensum
(dessverre ☺)*

- Anvendelser
 - Logistikk, fabrikk-planlegging og konstruksjon, produksjonsstyring, lagring og distribusjon av varer
 - Investeringsanalyse
 - Nettverksanalyse og protokoller
 - Trafikksimulering
 - Effekter av askeutslipp fra vulkaner og påvirkning av flytrafikk (NTNUs forrige superdatamaskin Njord)
 - Spredning av insekter og reduksjon av sprøytemidler i landbruket
 - Modellering av elkraftsystemer
 - Smart Grid: co-simulation der både kraftforsyningsnettverket og kommunikasjonsnettverket inngår
 - Modellering og analyse av digitale systemer (ELSYS)
 - Og mye mye mer!!!

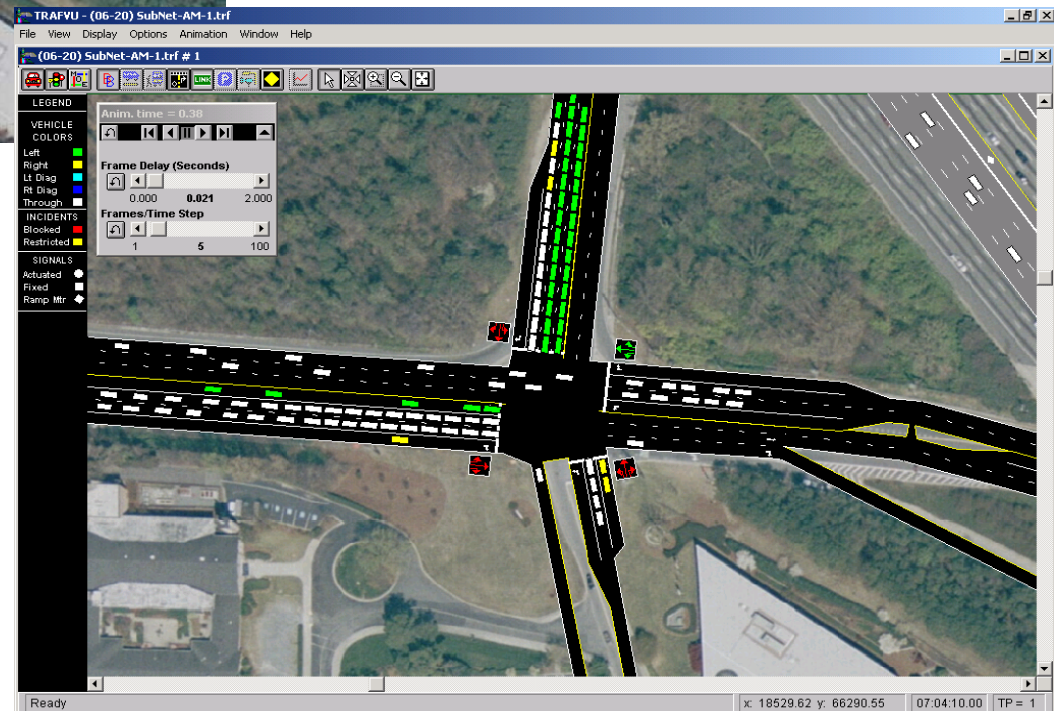
Trafikk-simulering

Ikke pensum



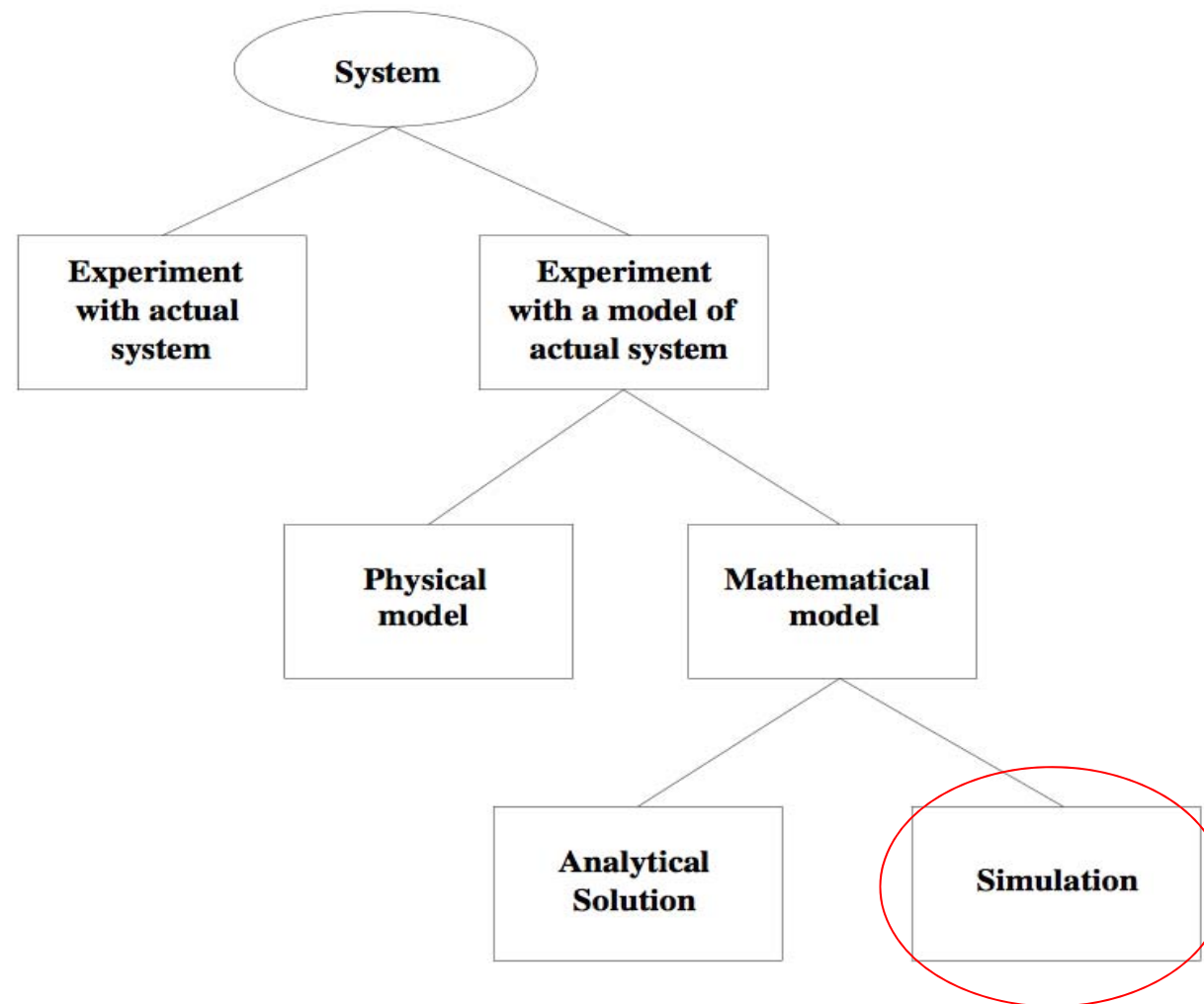
Aktuelt fysisk system og en realistisk simuleringsmodell av systemet

- Abstraksjon
 - Hva vil vi studere?
 - «Passelig mye forenklinger»



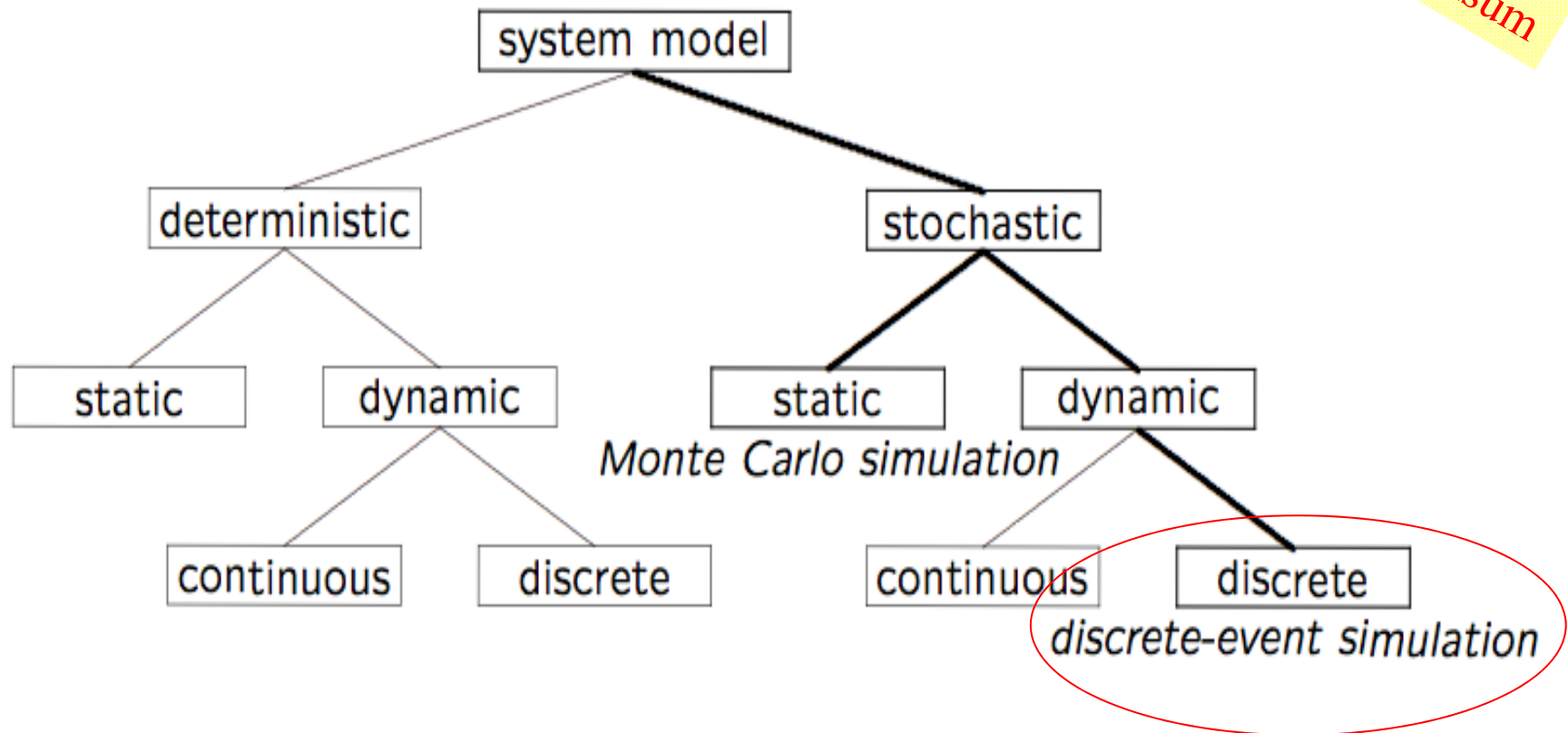
Simulering

Ikke pensum

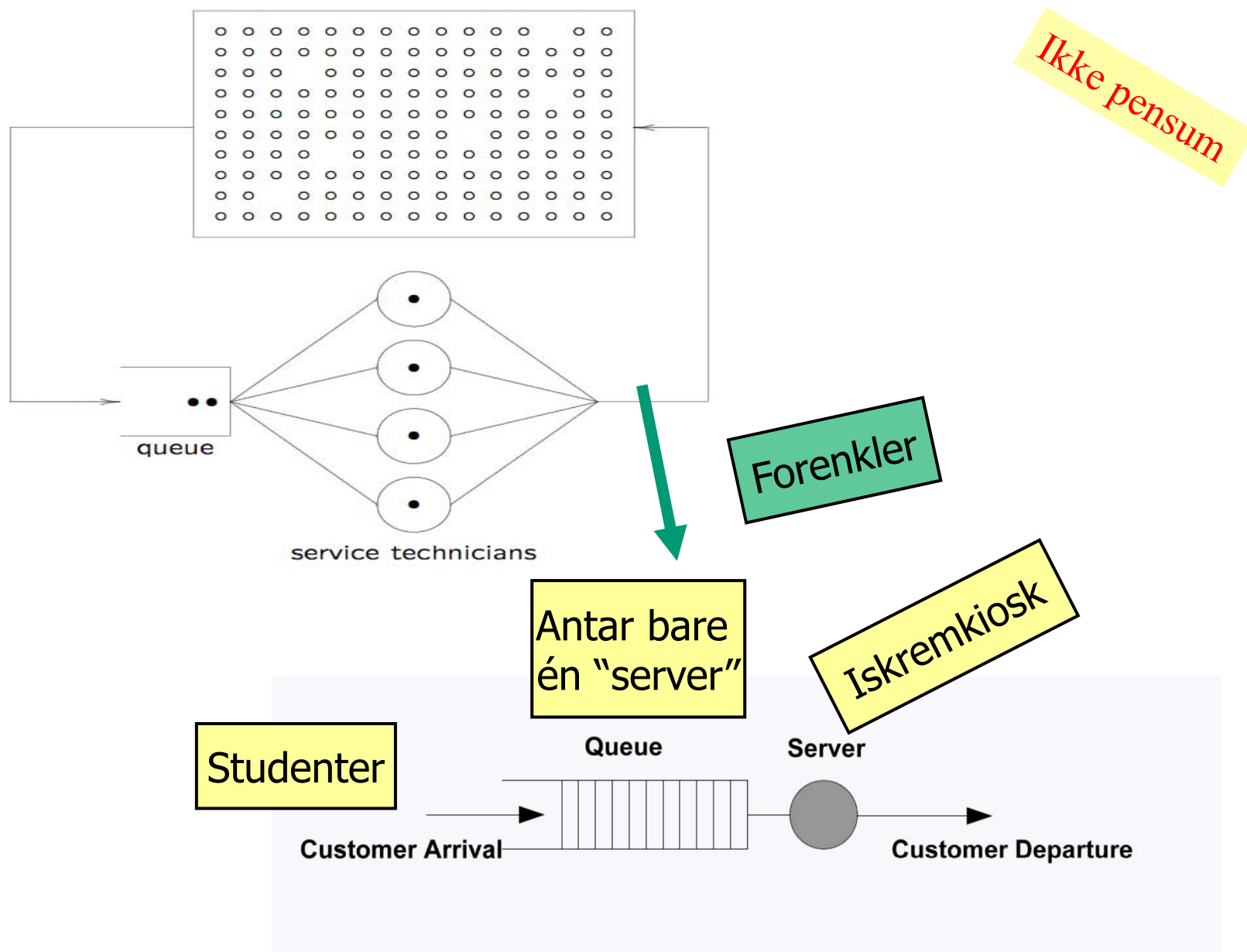


Simulering ...

Ikke pensum

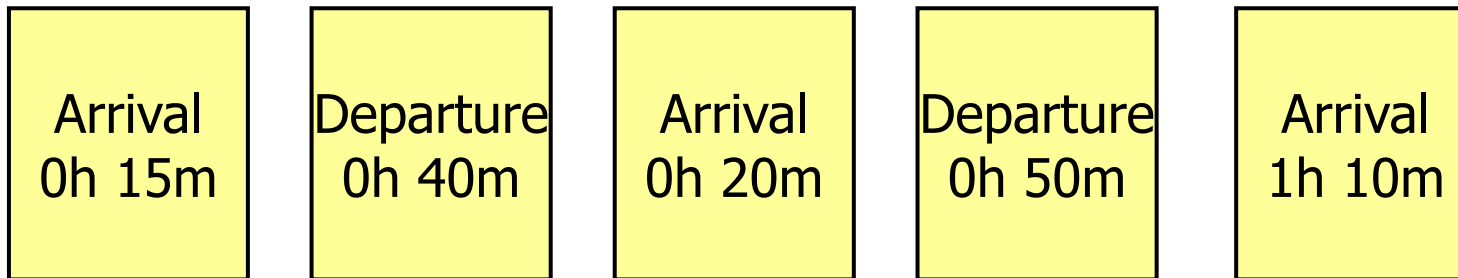


Iskremkiosk – tilstander/hendelser



Hendelser (Events)

Ikke pensum

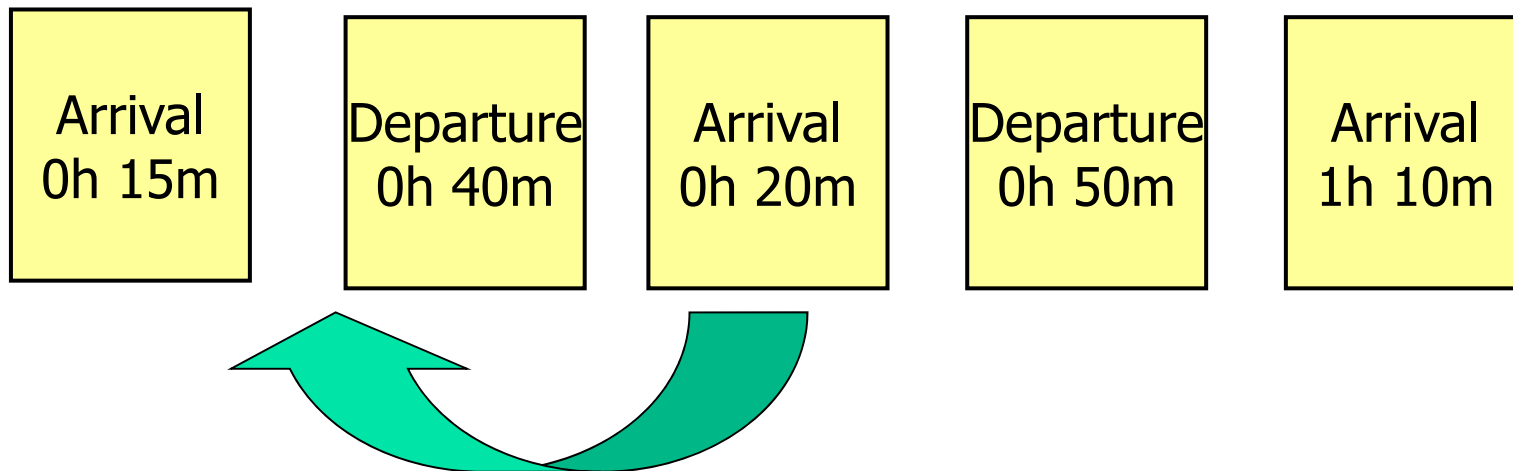


- Tenk på hendelser som Post-it© lapper som beskriver aktivitet som skal utføres ved ulik tid
- Hold dem i rekkefølge

Simulering, hendelser

Ikke pensum

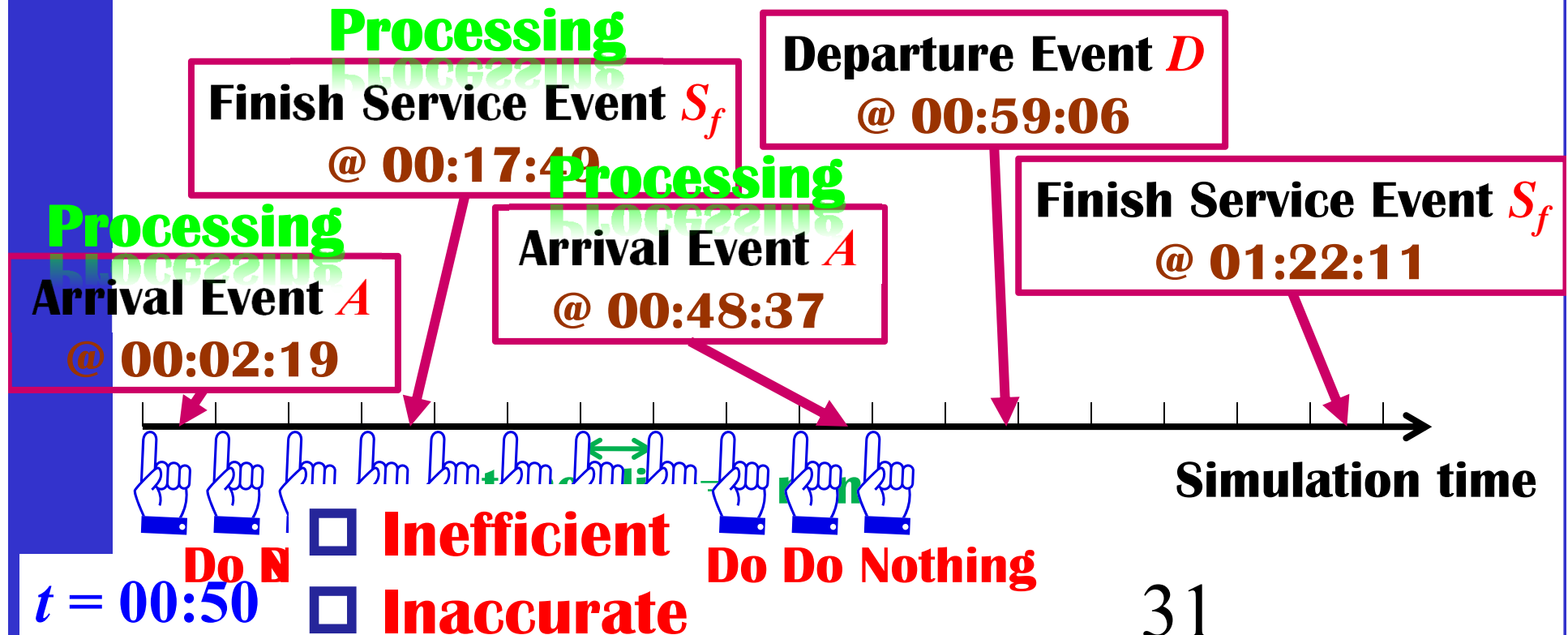
- Må utføres i riktig rekkefølge (simulerer systemet)



Time Handling

Ikke pensum

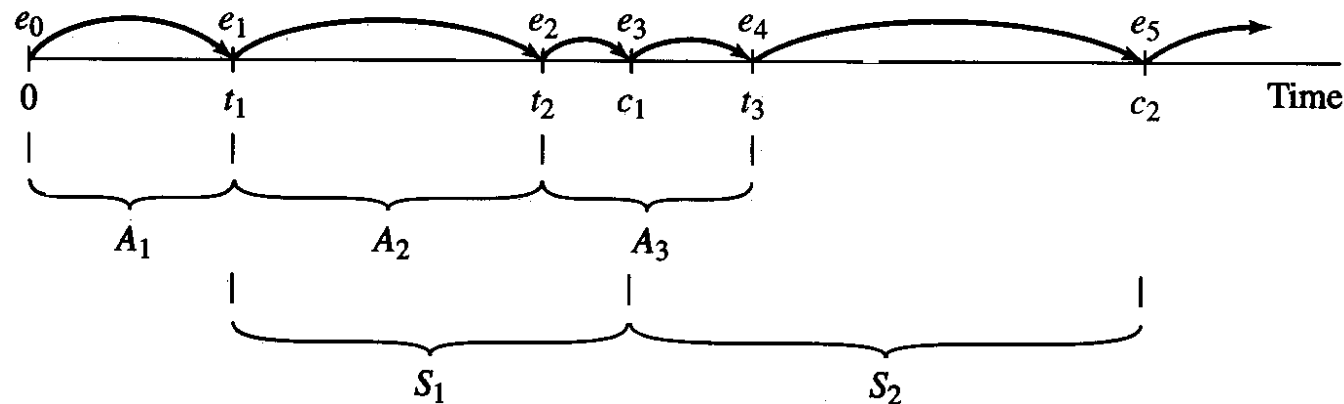
- How to progress Simulation time?
 - Time-slices Approach



Simulering

Ikke pensum

- «Discret event»
 - Hendelser (events) av interesse ved tid t_1, t_2, t_3 , osv.
 - Noterer/beregner/trekker «eksakt» når noe skjer
 - Håndterer en og en hendelse i rekkefølge
 - En hendelse gir gjerne en ny hendelse
- Figur (eksempel)
 - A_i er tid mellom Ankomster
 - S_i er Service-tid for kunde i osv.



Simulering av iskremkiosk, *episode 3*

- Nytt i dag 17/2
 - Bruk av <vector> av Events
 - Diskusjon rundt vector og sortering
 - Testing av program gjennom utskrift av eventVector
 - «litt mer dynamikk»
 - Innsetting av nye events
- Gjennomgang av diverse kode i Visual Studio
 - Gikk igjennom store deler av koden, men ikke i detalj på funksjonen simulateEvents (Den indiker retningen, og vil bli presentert i mer fullstendig utgave senere)



Iskrem_V3.zip