

Gjesteforelesning:

Minnehåndering - Fra skop til kopikonstruktør

TDT4102 — Prosedyre- og objektorientert programmering

Martin Tang Bruland, *Management Consultant* @ **Rystad Energy** (ex-vit.ass.)

18.02.2025



Martin Bruland

Management Consultant
Advisory

NTNU:

2019 – 2024: Siv.Ing. Nanotek

2020 – 2024: Stud.Ass, Und.Ass, Vit.Ass ved **IDI**

Annen tech-erfaring:

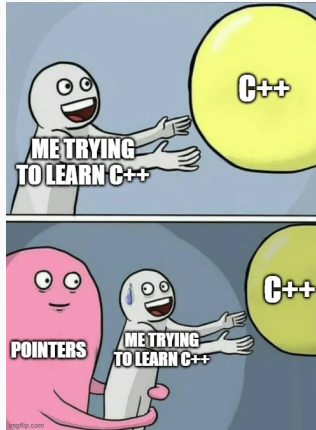
2021: Tech for gass-trading, **Equinor**

2023: Risikostyring, **NBIM** («Oljefondet»)

Nå:

I **Rystad Energy** hjelper jeg bedrifter å ta strategiske beslutninger

Hvorfor minne?



Hvorfor minne?

Minnekontroll er en av superkreftene til C++

(Mye mer fleksibilitet enn Python)



Hvorfor minne?

Noe av tematikken kan overføres til andre programmeringspråk

```
df_sorted = df ##### PROBLEM  
df_sorted = df_sorted.sort_index()  
  
df_sorted = df.copy() # Fikset  
df_sorted = df_sorted.sort_index()
```



```
int a = 5;
```



```
int* a = new int;
```

```
a* = 5;
```



Plan for dagen

Minnehåndtering i C++

- Stakk og heap

- Skop

- Minneadresser og pekere

- Heap

Pekere i objektorientert programmering

Hva slags minne?

-
-



RAM

- Alle variablene våre lagres i RAM (Random Access Memory)
-



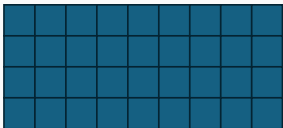
RAM

- Alle variablene våre lagres i RAM (Random Access Memory)
- Dette er midlertidig arbeidsminne som lånes ut til programmene når de er i bruk



RAM består av stakk og heap

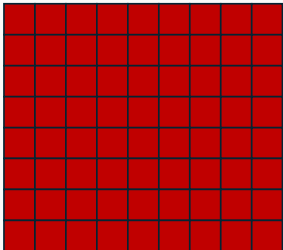
Stakk



- **Raskt**
- **Enkelt** (minne frigjøres automatisk)



Heap



- **Mer plass**
- **Dynamisk** (vi velger hvor lenge minnet skal holdes på)



Minidefinisjon: Et *skop* et er sett med krøllparanteser

```
{  
  // Dette er et skop  
}
```

Hvordan bruke stakk



- **Allokering** (= sette av minne):

```
int enVariabel = 5;
```

-

Hvordan bruke stakk



- **Allokering** (= sette av minne):
`int enVariabel = 5;`
- **Deallokering** (= frigjøre minnet igjen):
Skjer automatisk når variabelen går ut av *skop*.

Demo: Bruk av stakk

Oppsummering 1

- Vi kan lagre variabler på *stakk* og *heap*
- En variabel på **stakk** *dealloceres* når den går ut av *skop* (et sett med krøllparenteser)

Skop - Blokkskop

```
if (condition){  
    // Skop  
}
```

```
for (int i = 0; i < 5; i++){  
    // i er en del av skopet  
}
```

```
while(condition){  
    // Skop  
}
```

```
void enFunksjon(int x){  
    // x er en del av skopet  
}
```

```
switch(condition){  
    // Skop  
}
```

```
try{  
    // Et skop  
}  
catch(std::exception& e) {  
    // Et annet skop  
}
```

```
namespace n{  
    // Skop  
}  
  
class Klasse{  
    // Skop  
};
```

Disse skopene fungerer litt annerledes, og man bruker operatoren `::` for å hente medlemmene.

Spørsmål?

Demo: Skop

Hver plass i minnet har en **adresse**.



RAM består av stakk og heap



Demo: Minneadresser

Minneadresser og pekere

```
string enStreng = "Onsdager er kult!";
```

```
cout << &enStreng << endl;
```

```
// -> 0x16fdff114
```

```
cout << *(&enStreng) << endl;
```

```
// -> Onsdager er kult!
```



Minneadresser og pekere

```
string enStreng = "Onsdager er kult"
```

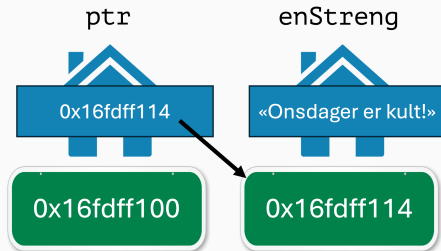
```
string* ptr = &enStreng;
```

```
// En peker er en variabel
```

```
// som holder en minneadresse
```

```
cout << *ptr << endl;
```

```
// -> Onsdager er kult!
```



Hvis vi vil at pekeren skal være "tom", setter vi den lik **nullpeker**

```
ptr = nullptr;
```

Fra stakk til heap



0x16fdff110



0x16fdff114



0x16fdff118



0x100c08150



0x100c08160



0x100c08160

Hvordan bruke heap

- **Allokering:**

`new int` setter av plass til en int og spytter ut adressen.

-

Eksempel:

```
int* intPtr = new int{3};  
// Setter av minne, gir det verdien 3, og lagrer adressen i en peker  
*intPtr = 5;  
0  
delete intPtr;  
0
```


Hvordan bruke heap

- Allokering:

`new int` setter av plass til en int og spytter ut adressen.

-

Eksempel:

```
int* intPtr = new int{3};
```

// Setter av minne, gir det verdien 3, og lagrer adressen i en peker



Hvordan bruke heap

- Allokering:

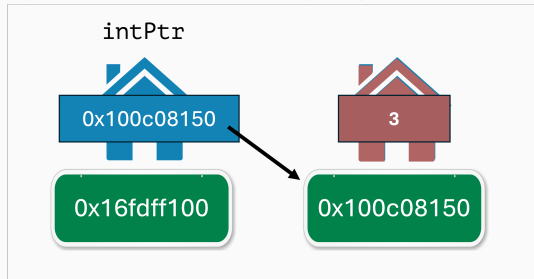
`new int` setter av plass til en int og spytter ut adressen.

-

Eksempel:

```
int* intPtr = new int{3};
```

// Setter av minne, gir det verdien 3, og lagrer adressen i en peker



Hvordan bruke heap

- **Allokering:**

`new int` setter av plass til en int og spytter ut adressen.

- **Deallokering:**

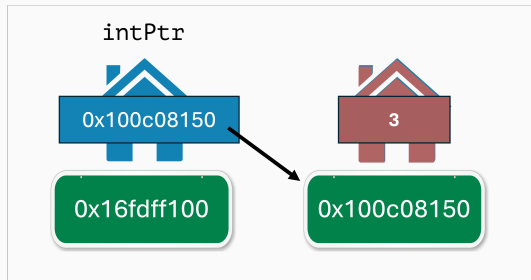
`delete` frigjør minnet på en adresse.

Eksempel:

```
int* intPtr = new int{3};  
// Setter av minne, gir det verdien 3, og lagrer adressen i en peker  
*intPtr = 5;  
// Vi endrer verdien til 5  
delete intPtr;  
// Når vi ikke trenger det lenger, frigjør vi minnet.
```

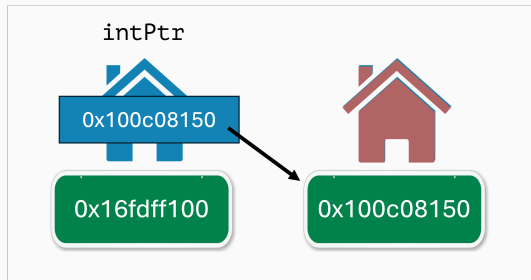
Hvordan bruke heap

```
int* intPtr = new int{3};  
// Setter av minne, gir det verdien 3, og lagrer adressen i en peker  
*intPtr = 5;  
// Vi endrer verdien til 5  
delete intPtr;  
// Når vi ikke trenger det lenger, frigjør vi minnet.
```



Hvordan bruke heap

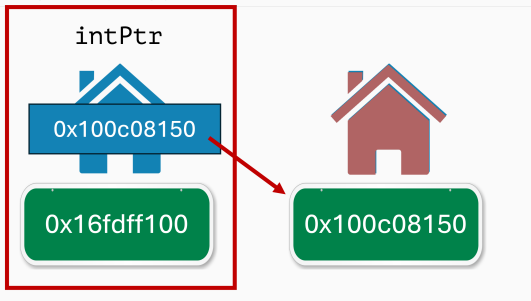
```
int* intPtr = new int{3};  
// Setter av minne, gir det verdien 3, og lagrer adressen i en peker  
*intPtr = 5;  
// Vi endrer verdien til 5  
delete intPtr;  
// Når vi ikke trenger det lenger, frigjør vi minnet.
```



Hvordan bruke heap

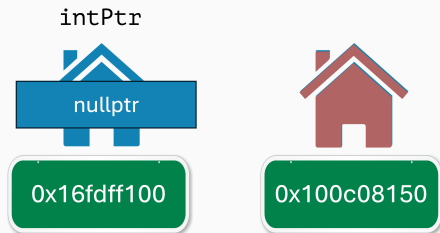
```
int* intPtr = new int{3};  
// Setter av minne, gir det verdien 3, og lagrer adressen i en peker  
*intPtr = 5;  
// Vi endrer verdien til 5  
delete intPtr;  
// Når vi ikke trenger det lenger, frigjør vi minnet.
```

NB!!!!
Dangling pointer



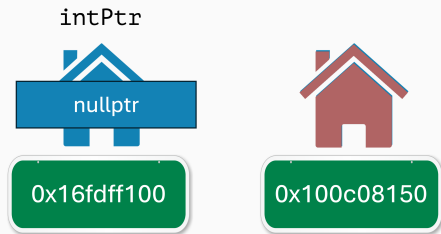
Hvordan bruke heap

```
int* intPtr = new int{3};  
// Setter av minne, gir det verdien 3, og lagrer adressen i en peker  
*intPtr = 5;  
// Vi endrer verdien til 5  
delete intPtr;  
// Når vi ikke trenger det lenger, frigjør vi minnet.
```



Hvordan bruke heap

```
int* intPtr = new int{3};  
// Setter av minne, gir det verdien 3, og lagrer adressen i en peker  
*intPtr = 5;  
// Vi endrer verdien til 5  
delete intPtr;  
// Når vi ikke trenger det lenger, frigjør vi minnet.  
intPtr = nullptr;  
// NB: Etter vi bruker delete, bør vi sette pekeren til nullpeker
```



Hvordan bruke heap

Vi kan også allokere til flere adresser samtidig

```
char* p = new char[5];  
// Setter av plass til 5 chars
```

```
p[3] = 'A';  
// Endrer på den fjerde char'en
```

```
delete[] p;  
// Frigjør de 5 plassene.
```

Dette kalles en *array* (men det er enklere å bruke *vector*)

Dynamisk minneallokering

Å allokere minne med `new` kalles gjerne *dynamisk minneallokering*.

Det er fordi vi klarer å sette av en mengde minne som ikke er kjent på forhånd (under kompilering).

```
cout << "Hvor mye int skal vi ha plass til?" << endl;
```

```
int n;
```

```
cin >> n;
```

```
int* ptr = new int[n];
```

- Operatoren `&` henter en minneadresse
- Operatoren `*` gjør det motsatte
- En minneadresse kan lagres i en variabel av type *peker*, f.eks. `char*`
- Vi kan allokere og deallokere på heap gjennom `new` og `delete`



1. Minnelekkasje

```
for(int i = 0; i < 10; i++) {  
    std::string* helpMessage = new std::string("Oh no!");  
    std::cout << *helpMessage << std::endl;  
}
```

2. Dangling pointer

```
CardDeck* ptr = new CardDeck;  
delete ptr;
```

// ptr peker fremdeles til samme sted

*// Hvis vi skriver *ptr, vil vi fremdeles lese hva som står der i minnet*

3. Double free

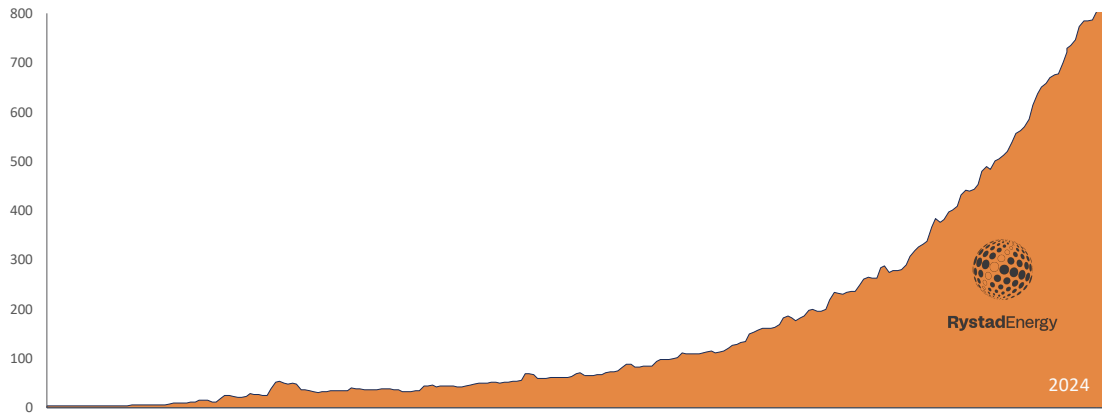
```
CardDeck* ptr = new CardDeck;  
delete ptr;
```

```
delete ptr; // FEIL: Vi prøver å deallokere et minnne som ikke er allokert
```

Spørsmål?

Fra et par konsulenter, til globalt ledende kunnskapshus på 20 år

Rystad Energy employees
of FTEs



RystadEnergy

2024

Putting the pieces together – what does it mean?



We collect data at the **lowest level of granularity** to create a complete and detailed analysis

Assets & projects are the building blocks:



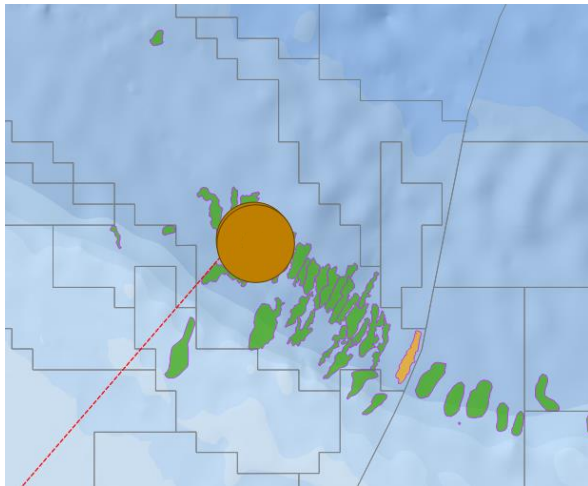
Asset Name	Hornsea Three, GB	Developer	Orsted
Country	United Kingdom	Foundation Concept	Monopile
Capacity (MW)	2850	Distance to Shore (km)	121.0
Start up Year	2027	Water Depth (meter)	48.5
Plant Status	Approved	Area (sq. km)	284
Number of Turbines	190	Latitude	53.87

Assets & projects are the building blocks:



Name		Noor Abu Dhabi-Sweihan	Solar PV
Location		Status	Operating
Country		UAE	
Province		Abu Dhabi	
Key dates		Plant capacity	
Approval	5/18/2017	MW AC	935.00
Financial close	11/3/2017	Lifetime capacity factor	31.0%
Construction	12/12/2017		
Start up	7/1/2019		
Main owner	Abu Dhabi Power Corp;60;Jinko;20;Marubeni;20		
Developer	Marubeni Corporation; Jinko Solar		
Technical information			
Main Contractor EPCI	Sterling & Wilson	Number of solar panels	3200000
O&M Provider	Sterling & Wilson	Inverter Manufacturer	Ingeteam
Solar Panel Manufacturer	Jinko	Tracking	Fixed
Panel Technology	Mono-BSF	Tracking Manufacturer	

Assets & projects are the building blocks:

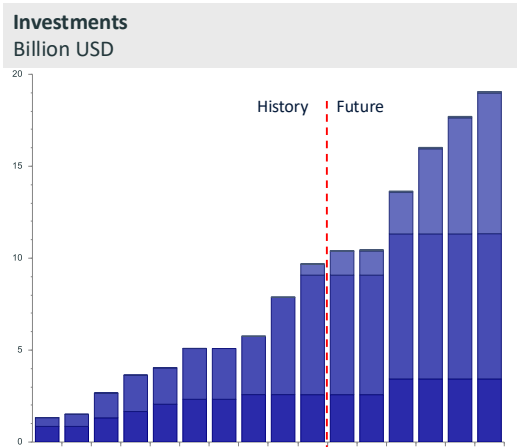
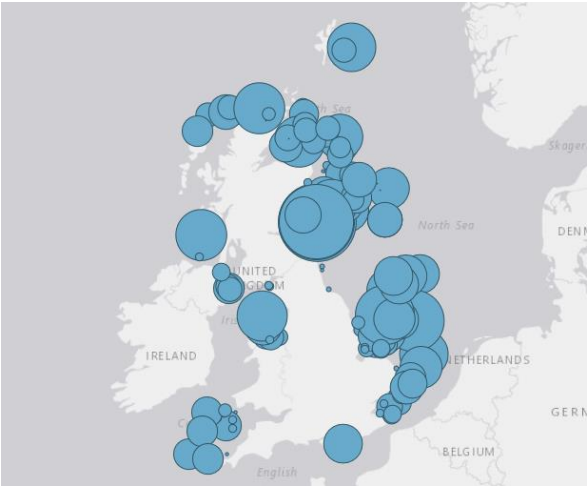


Asset	Liza Phase 2 (Unity)	Liza Phase 1 (Destiny)
Discovery Year	2015	2015
Approval Year	2019	2017
Start-up Year	2022	2019
LifeCycle Category	Producing	Producing
Facility Detail	FPSO	FPSO
Original Resources, Mmboe	556	514
Remaining Resources, Mmboe	502	397

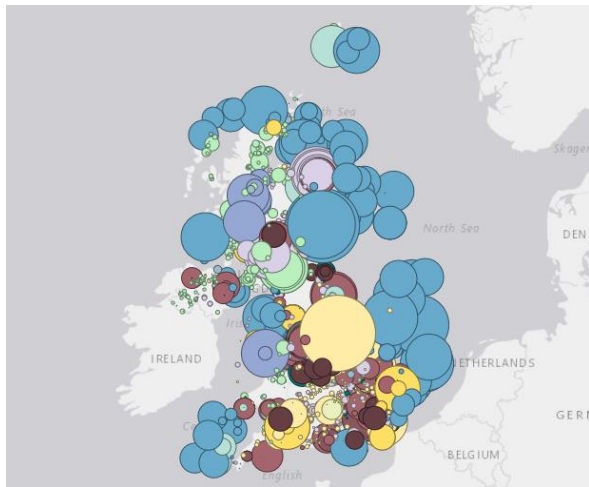
Assets & projects are the building blocks to this approach:



Aggregating to see patterns for the future – with different use-cases

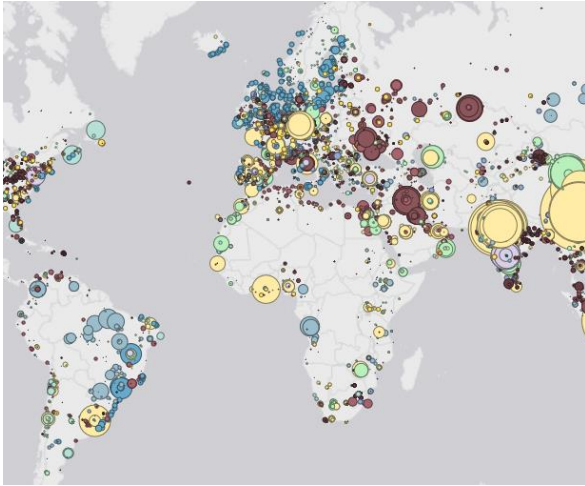


Consistent, complete and integrated approach to Energy Intelligence

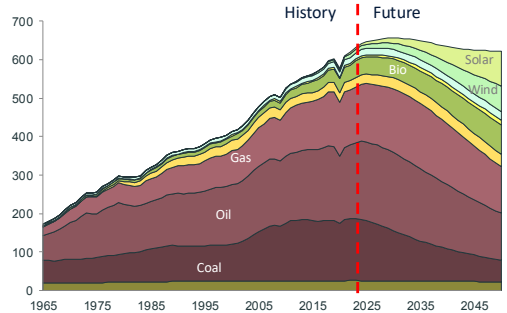


Solar.
Wind.
Hydrogen.
CCUS.
Batteries.
Geothermal.
Power.
Oil & Gas.
Emissions.
Supply Chain.

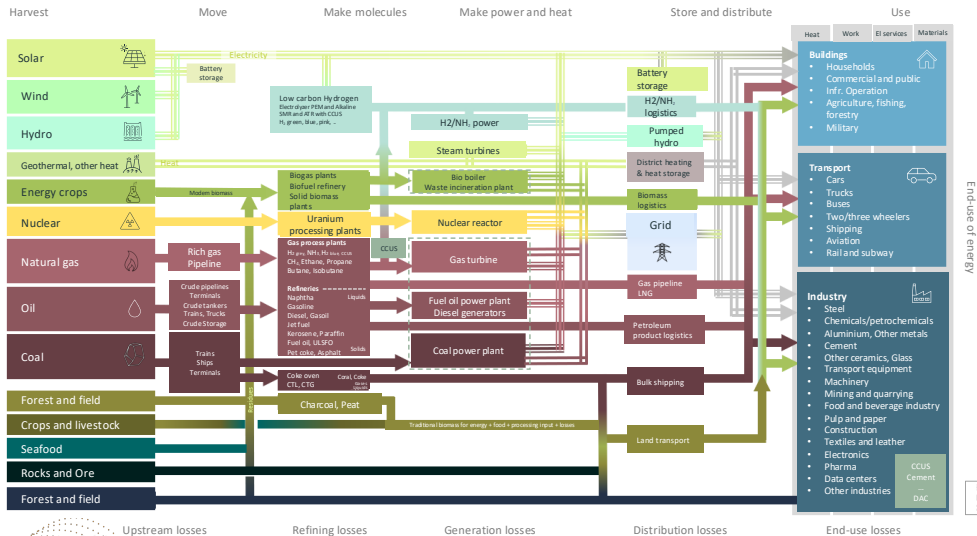
Or even modelling future **global energy scenarios**



Primary Energy Supply by source
Exajoules



Our energy map



Programming can be useful across different departments



Technology

Building the infrastructure and interfaces



Analysis

Finding market trends, publishing reports, building products



Advisory

Analysis as a piece of the bigger picture

So far, I have used programming for



Advanced Modeling

Predicting oil and product prices
-> Published as a whitepaper



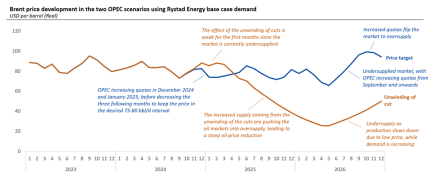
Power Market

Using APIs for automating data collection on the Power market



Gas Market

Analyzing gas market transactions



Other projects I have worked on these 6 months



Subsea

Subsea market projection



M&A

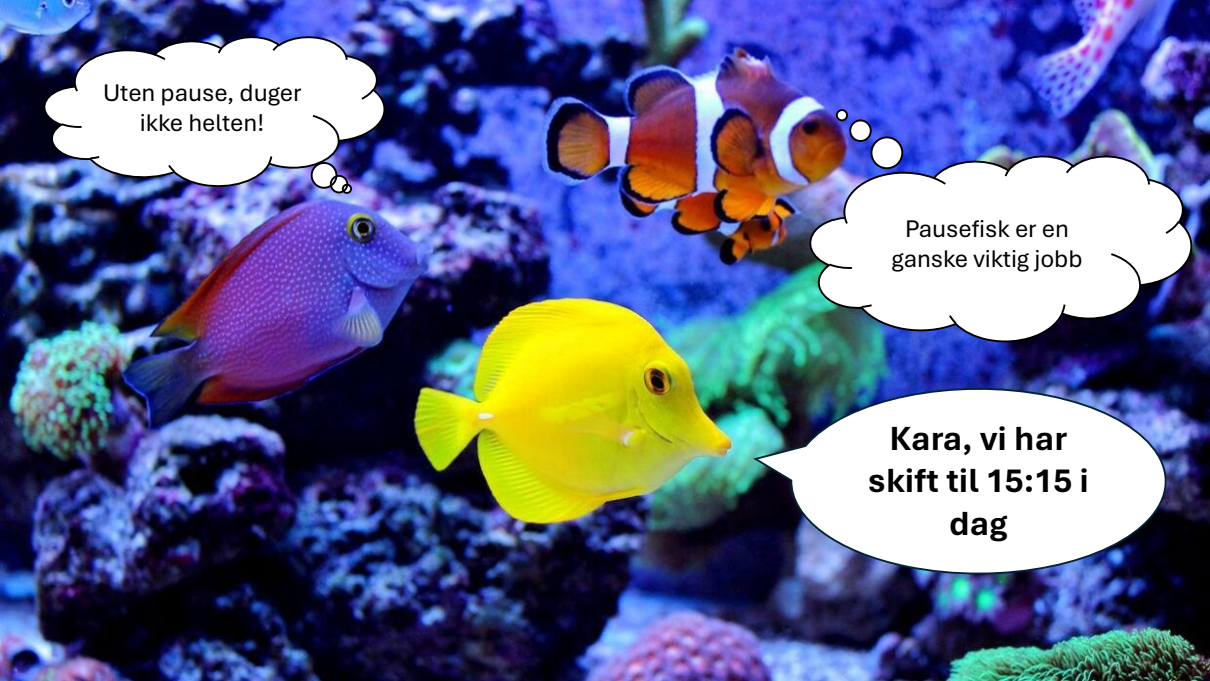
Helping a PE fund analyze an acquisition target



Market Entry Strategy

Helping an engineering firm explore a new market

Takeaway: Programmering er blant de aller viktigste emnene jeg har tatt



Uten pause, duger
ikke helten!

Pausefisk er en
ganske viktig jobb

**Kara, vi har
skift til 15:15 i
dag**

Minnehåndtering i C++

Pekere i objektorientert programmering

Destruktør

Kopikonstruktør

Tilordningsoperator

The impact of classroom seating location and computer use on student academic performance

Paris Will *, Walter F. Bischof, Alan Kingstone

Department of Psychology, University of British Columbia, Vancouver, BC, Canada

* paris.will.19@ucl.ac.uk

Abstract

A student's ability to learn effectively in a classroom setting is subject to many factors. While some factors are difficult to regulate, this study explores two factors that a student, or instructor, has full control over, namely 1) seating position, and 2) computer usage. Both factors have been studied considerably with regard to their effects on student performance,

Fremme i salen → bedre resultater*

The Surprising Impact of Seat Location on Student Performance

Katherine K. Perkins and Carl E. Wieman, University of Colorado at Boulder, Boulder, CO

Every physics instructor knows that the most engaged and successful students tend to sit at the front of the class and the weakest students tend to sit at the back. However, it is normally assumed that this is merely an indication of the respective seat location preferences of weaker and stronger students. Here we present evidence suggesting that in fact this may be mixing up the cause

of large physics lecture halls that need to be further explored.

The Course

This study was done in the "Physics of Everyday Life" course we taught at the University of Colorado in Boulder. This is an algebra-based introductory physics course for nonscience, nonengineering majors

13
1000-0000/2018/10-0000-00



Contents lists available at ScienceDirect

Learning and Instruction

journal homepage: www.elsevier.com/locate/learninstruc



Do students learn better when seated close to the teacher? A virtual classroom study considering individual levels of inattention and hyperactivity-impulsivity

Friederike Blume^{a,b,c}, Richard Göllner^{b,c}, Korbinian Moeller^{a,b,d}, Thomas Dresler^{a,b}, Ann-Christine Ehlig^{a,b}, Caterina Gawrilow^{a,b,e}

^aDepartment of Psychology, University of Tübingen, Schleichersstr. 4, 72076, Tübingen, Germany

^bLEARN Graduate School & Research Network, University of Tübingen, Weber-Strasse/Strasse 12, 72076, Tübingen, Germany

^cMax Planck Institute of Education Sciences and Psychology, University of Tübingen, Konigsplatz 6, 72076, Tübingen, Germany

^dLudwig-Maximilians-Universität München, Schellingstr. 6, 80539, München, Germany

^eDepartment of Psychiatry and Psychotherapy, University Hospital Tübingen, Calwerstrasse 14, 72076, Tübingen, Germany

^fCenter for Individual Development and Adaptive Education of Children at Risk (IDeA), Deutscher Institut für Internationale Pädagogische Forschung (DIPF), Bratscher Strasse 6, 60325, Frankfurt am Main, Germany

ARTICLE INFO

Keywords:
Virtual reality
ADHD symptoms
Learning
School

ABSTRACT

This study investigated whether students in grades 5 and 6 learned better when seated proximal to the teacher during a virtual classroom math lesson, taking individual levels of inattention and hyperactivity-impulsivity (i.e., ADHD symptoms) into account. In general, students learned better in the proximal seat location compared to a distal one. Additionally, more intense symptoms levels required learning more. When considering individual levels of ADHD symptoms, students' learning outcomes did not specifically benefit from a proximal seat location. Consequently, the present study did not support the general assumption that a proximal seat location fosters academic achievement in students experiencing individual levels of inattention and hyperactivity-impulsivity.

doi: <https://doi.org/10.1119/1.1845987>

doi: <https://doi.org/10.1016/j.learninstruc.2018.10.004>

*(men det finnes også sprikende funn)

The impact of classroom seating location and computer use on student academic performance

Paris Will *, Walter F. Bischof, Alan Kingstone

Department of Psychology, University of British Columbia, Vancouver, BC, Canada


* paris.will.19@ucl.ac.uk

Abstract

A student's ability to learn effectively in a classroom setting is subject to many factors. While some factors are difficult to regulate, this study explores two factors that a student, or instructor, has full control over, namely 1) seating position, and 2) computer usage. Both factors have been studied considerably with regard to their effects on student performance,

PC i forelesning → dårligere resultater for studenter som sitter bak deg

Computers & Education 62 (2013) 24–31




ELSEVIER

Contents lists available at [SciVerse ScienceDirect](#)

Computers & Education

journal homepage: www.elsevier.com/locate/compedu



Laptop multitasking hinders classroom learning for both users and nearby peers

Faria Sana^a, Tina Weston^{b,c}, Nicholas J. Cepeda^{b,c,*}

^aMcMaster University, Department of Psychology, Neuroscience, & Behaviour, 1280 Main Street West, Hamilton, ON L8S 4K1, Canada
^bYork University, Department of Psychology, 4700 Keele Street, Toronto, ON M3J 1P3, Canada
^cYork University, LaMarsh Centre for Child and Youth Research, 4700 Keele Street, Toronto, ON M3J 1P3, Canada

doi: <https://doi.org/10.1016/j.compedu.2012.10.003>

Research Article



The Pen Is Mightier Than the Keyboard: Advantages of Longhand Over Laptop Note Taking



Pam A. Mueller¹ and Daniel M. Oppenheimer²

¹Princeton University and ²University of California, Los Angeles

Psychological Science
2014, Vol. 25(6) 1159–1168
© The Author(s) 2014
Reprints and permissions:
sagepub.com/journalsPermissions.nav
DOI: 10.1177/0956797614524581
pss.sagepub.com



doi: <https://doi.org/10.1177/0956797614524581>

Case for økten: Vektor-klassen er ikke oppfunnet enda. Vi skal lage en klasse `IntVector`, som kan brukes til å oppbevare heltall.

RAII (Resource Aquisition is Initialization)

1. Minneallokering i konstruktøren
- 2.

IntVektor: Det vi har laget så langt

```
class IntVector{  
    private:  
        int size;  
        int* content;  
    public:  
        IntVector(int n): size{n}, content{new int[n]}{  
            cout << "Allocating memory" << endl;  
        }  
        void print() const;  
        void set(int index, int value);  
};
```

Utfordringer med denne klassen?

Vi allokere minne i konstruktøren, men frigjør det aldri.

IntVektor: Det vi har laget så langt

```
class IntVector{  
    private:  
        int size;  
        int* content;  
    public:  
        IntVector(int n): size{n}, content{new int[n]}{}  
        void print() const;  
        void set(int index, int value);  
};
```

Utfordringer med denne klassen?

Vi allokerer minne i konstruktøren, men frigjør det aldri. Minnelekkasje

DESTRUCTOR



Destruktøren

Destruktøren er en medlemsfunksjon som kalles når en variabel *går ut av skop*:

```
{  
    IntVector v = IntVector(5); // Allokering  
} // Destruktøren kalles på
```

```
IntVector* ptr = new IntVector(5);
```

Destruktøren

Destruktøren er en medlemsfunksjon som kalles når en variabel *går ut av skop*:

```
{  
    IntVector v = IntVector(5); // Allokering  
} // Destruktøren kalles på
```

PS: Når man bruker **delete** på en peker, vil konstruktøren til objektet som ligger der kalles på:

```
IntVector* ptr = new IntVector(5);  
delete prt; // Destruktøren til IntVector kalles
```

RAII (Resource Aquisition is Initialization)

1. Minneallokering i konstruktøren
- 2.

RAII (Resource Aquisition is Initialization)

1. Minneallokering i konstruktøren
2. Deallokering i destruktøren

Implementeres på følgende måte:

```
class IntVector{  
    ...  
    IntVector(int n); // Konstruktor  
    ~IntVector(); // Destruktør  
};
```

Implementeres på følgende måte:

```
class IntVector{  
    ...  
    IntVector(int n); // Konstruktør  
    ~IntVector(){  
        // Her må vi deallokere minne  
    }  
};
```


Tilbake til koden

Vi fikk problemer da vi gjorde følgende:

```
int main(){  
    IntVector v1{5};  
  
    IntVector v2 = v1; // Kaller på kopikonstruktør  
    IntVector v3{v1};  // Kaller på kopikonstruktør  
  
}
```

Vi har ikke laget noe kopikonstruktør?

Vi har ikke laget noe kopikonstruktør?

Da gjør C++ det for oss.

Vi har ikke laget noe kopikonstruktør?

Da gjør C++ det for oss.

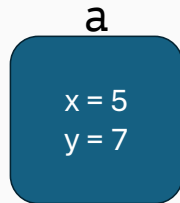
Og vanligvis går det greit. Men ikke når vi har pekere i klassen.

Hva gjør denne standard-kopikonstruktøren?

```
class Point{  
    double x;  
    double y;  
};  
  
int main(){  
    Point a{5, 7};  
    Point b = a;  
}
```

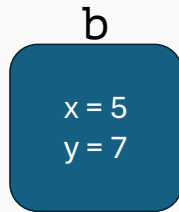
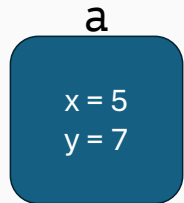
Hva gjør denne standard-kopikonstruktøren?

```
class Point{  
    double x;  
    double y;  
};  
  
int main(){  
    Point a5, 7;  
    Point b = a;  
}
```



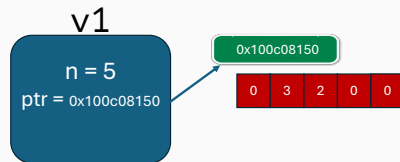
Hva gjør denne standard-kopikonstruktøren?

```
class Point{  
    double x;  
    double y;  
};  
  
int main(){  
    Point a{5, 7};  
    Point b = a;  
}
```

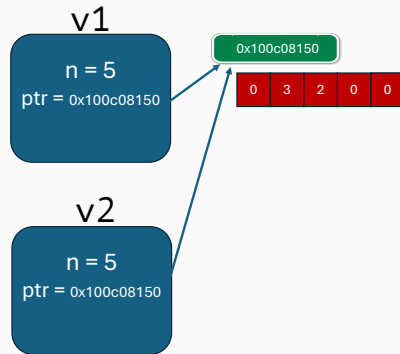



```
int main(){  
    IntVector v1{5};  
    ... // Plasserer noen verdier inn  
    IntVector v2{v1}; // Lager kopi  
}
```

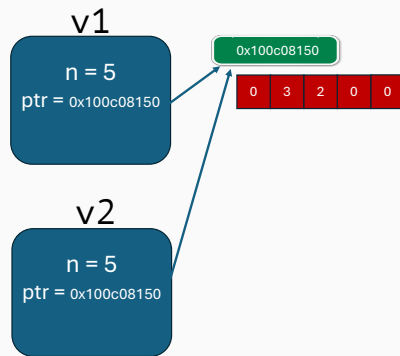
```
int main(){  
    IntVector v1{5};  
    ... // Plasserer noen verdier inn  
    IntVector v2{v1}; // Lager kopi  
}
```



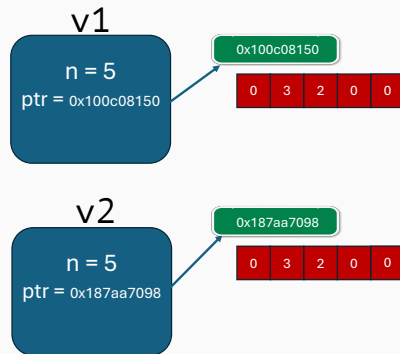
```
int main(){  
    IntVector v1{5};  
    ... // Plasserer noen verdier inn  
    IntVector v2{v1}; // Lager kopi  
}
```



Dette kalles **grunn kopiering**



Vi ønsker heller **dyp kopiering**



Dette var feilen jeg gjorde i Python

```
df_sorted = df ##### PROBLEM  
df_sorted = df_sorted.sort_index()  
  
df_sorted = df.copy() # Fikset  
df_sorted = df_sorted.sort_index()
```

Hvordan overskriver vi kopikonstruktøren?

Hvordan overskriver vi kopikonstruktøren?

```
class IntVector{  
    ...  
    IntVector(const Intvector& v){  
        // Overstyrer kopikonstruktøren  
    }  
};
```

En konstruktør som tar inn et annet objekt av klassen.

Hvordan overskriver vi kopikonstruktøren?

```
class IntVector{  
    ...  
    IntVector(const Intvector& v){  
        // Overstyrer kopikonstruktøren  
    }  
};
```

En konstruktør som tar inn et annet objekt av klassen.

NB: Vi må ta objektet inn *by reference*, hvorfor?

Tilbake til koden

Tilordningsoperator

```
int main(){  
    IntVector v1{5};  
    IntVector v2{v1}; // Kopikonstruktør, den har vi fikset  
  
    v2 = v1; // Problem  
    // Løsning: Oversikriv tilordningsoperatoren på samme måte  
}
```

Tilordningsoperator

```
class IntVector{
private:
    int size;
    int* content;
public:
    ...
    IntVector& operator=(const IntVector& rhs){
        size = rhs.size;
        delete content;
        contents = new int[size];

        for (int i = 0; i < size; i++){
            contents[i] = rhs.contents[i];
        }
    }
};
```

Når vi har dynamisk allokerede medlemmer i en klasse, bør vi som regel

1. Deallokere i destruktøren
2. Overskrive kopikonstruktøren (dyp kopiering)
3. Overskrive tilordningsoperatoren

Tips: Forenklet tilordningsoperator

Hvis vi først overskriver kopikonstruktøren, finnes det et triks for å overskrive tilordningsoperatoren:

```
class IntVector{
private:
    int size;
    int* content;
public:
    ...
    IntVector& operator=(IntVector other) { // Lager other
        // gjennom kopikonstruktør (by-value) son vi allerede fikset
        swap(*this, other);
        return *this; // this referer her til objektet
        //på venstre side av operatoren
    }
};
```