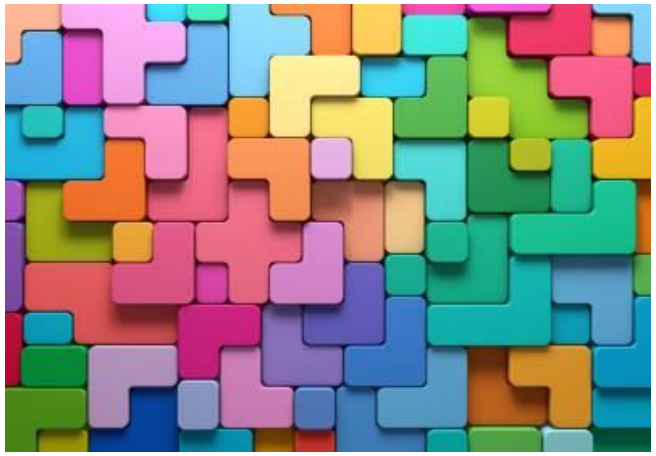
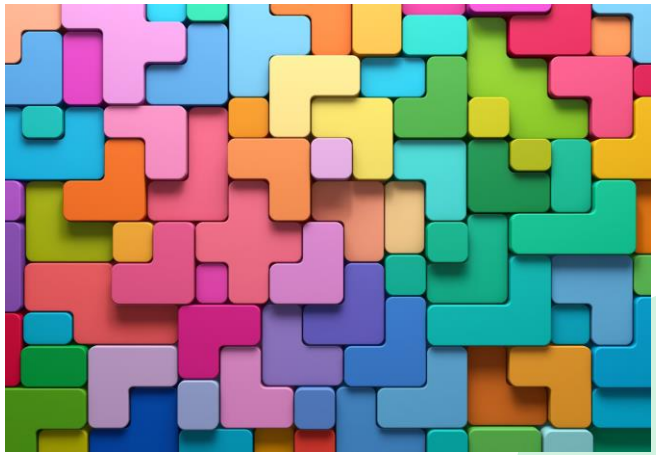
# REFERANSER OG SORTERINGSALGORITMER

FORELESNING 14

FREDAG 11/10

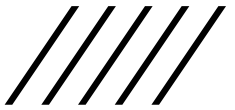




(bilder generert av bing image creator)

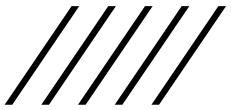
# **make** i Windows kan lage litt problemer

- Husk at **make** er frivillig å bruke – du kan kompilere normalt
- Hvis du likevel vil bruke **make** på Windows:
  - Installasjon: **conda install conda-forge::make**
    (i terminalvindu åpnet av VS Code)
  - Kompiler alltid med **-static-libstdc++** (legg til i Makefile)
  - Om nødvendig legg til
    **"C_Cpp.default.compilerPath":**
    **"C:\\msys64\\ucrt64\\bin\\g++.exe"**
    i VS Code sin **settings.json**



# Viktig forskjell på forelesninger og støttelitteratur

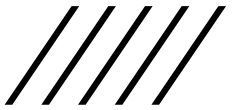
- I støttelitteraturen settes standardverdier på instansvariabler der disse defineres (for oss ville det vært i header-filen)
- I forelesning (livekoding) forrige uke gjorde vi ikke det, men lot konstruktøren sette verdien
- Fordel: Standardverdiene (en del av implementasjonen) er dermed ikke synlige i grensesnittet
- Ulempe: Må passe på å sette verdiene i alle konstruktørene
- Velg én av disse 2 måtene – ikke bland dem!



# LIVEKODING: SMARTE PEKERE

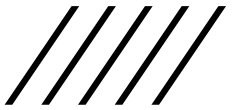
# Rettelse: Forskjell på **struct** og **class**

- Det kan være smart å bruke **struct** når vi bare trenger dataobjekter uten metoder (men de kan ha metoder også)
- Alt i en **struct** er **public** som standard
- Alt i en **class** er **private** som standard (hvis ikke vi spesifiserer at det skal være **public**)



# Er arrays og pekere samme ting?

- Nei, det er viktige forskjeller (f.eks. **sizeof**)
- Men de bruker indeksering helt likt
- **en_array[2]** returnerer i teorien tredje element i en array
- **en_peker[2]** hopper 2 steg (steglengde i bytes avhenger av type) videre i minnet og tolker hva enn som ligger der som en verdi av typen til pekeren
- Når vi bruke **int* _data = new int[_capacity]** lager vi en peker til første element i en array (men indeksering vil altså funke)



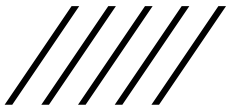
# LIVEKODING: REFERANSER

# Hva betyr int& for noe?

| Symbol | Etter type | Før variabelnavn |
|---|---|---|
| (ikke noe symbol) | Variabel (heltall)<br>**int x = ...** | Verdi<br>**... = x;** |
| * | Peker (minneadresse)<br>**int* x = ...** | Verdi på minneadressen<br>**... = *x** |
| & | Referanse<br>**int& x = ...** | Minneadresse<br>**... = &x** |

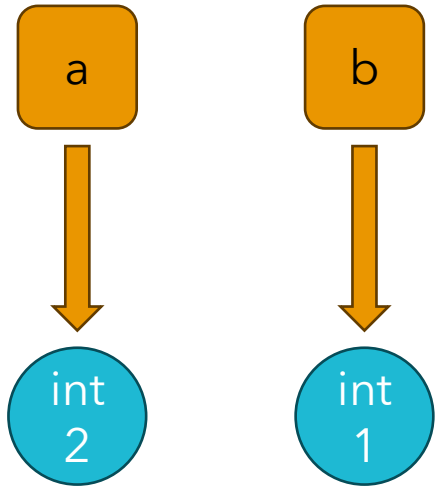
Feilmelding hvis x ikke er en peker

- **int y = x** *kopierer* verdien til et annet sted i minnet (&y != &x)
- **int& y = x** gjør at x og y peker på samme verdi (&y == &x)



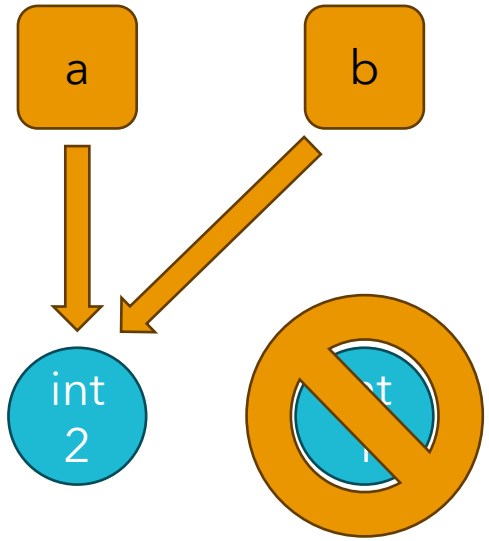
# Vanlige variabler

| C++ | Python | Verdi av a | Verdi av b |
|---|---|---|---|
| int a = 1; | a = 1 | 1 | |
| int b = a; | b = a | 1 | 1 |
| a = 2; | a = 2 | 2 | 1 |



# C++: To måter å overskrive (tilordne) flere variabler samtidig på

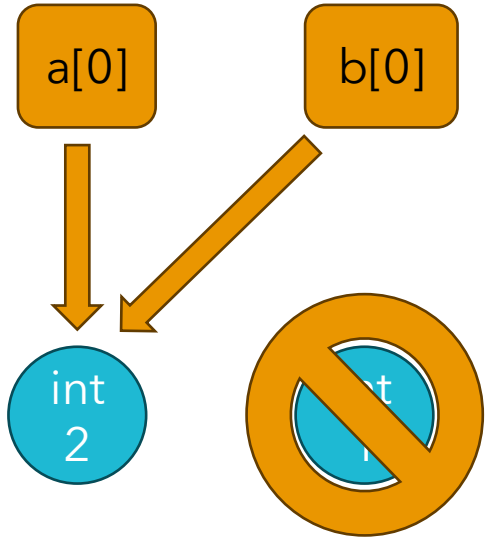
| Med peker | Med referanse | Verdi av a | Verdi av *b/b |
|---|---|---|---|
| int a = 1; | int a = 1; | 1 | |
| int* b = &a; | int& b = a; | 1 | 1 |
| a = 2; | a = 2; | 2 | 2 |



# Men det gjelder overskriving av hele objektet, ikke *endring* av (muterbare) objekter

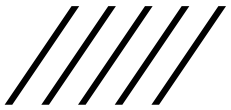
Må bruke peker for at dette skal funke i C++

| C++ | Python | Verdi av a[0] | Verdi av b[0] |
|---|---|---|---|
| int a[] = {1}; | a = [1] | 1 | |
| int b* = a; | b = a | 1 | 1 |
| a[0] = 2; | a[0] = 2; | 2 | 2 |



# Boblesortering (bubble sort)

6 5 3 1 8 7 2 4



# LIVEKODING: BUBBLE SORT

# Analyse av bubble sort

Let us analyze the cost of this algorithm. We first iterate through the whole list, after which the biggest element will be last. This means that the subsequent iteration can now be done through the whole list except the last element. Similarly, every new iteration requires going to one less element than the last one. For each iteration, we perform at least one comparison and perhaps a swap. Suppose that, for each iteration, we do $c$ operations. This gives us the sum

$$(n-1)c + (n-2)c + (n-3)c + \ldots + 2c + c.$$

This sum represents essentially a triangle number, which can be expressed as $\frac{n(n-1)}{2}c$.

As before, the coefficients and lower-order terms are unimportant, so the complexity analysis can be summarized by saying that bubble sort takes $\mathcal{O}(n^2)$ operations. Doubling the length of the input list makes bubble sort take roughly four times longer to sort the list.

Note that the algorithm is $\mathcal{O}(n^2)$ in the worst case and in the best case. Indeed, even if we are not doing any swaps, we still perform $\mathcal{O}(n^2)$ comparisons. This means that even in the best case in which we start with an *already sorted list*, bubble sort will spend $\mathcal{O}(n^2)$ operations to "sort" it. This can be improved.

# LIVEKODING: BUBBLE SORT

# (DEL 2)

# Analyse av forbedret bubble sort

In the worst-case scenario, we still need to carry out all the same iterations as before, so the worst case has not improved with this new algorithm. Fortunately, this worst-case scenario did not get any worse either; We added a few constant steps into the algorithm, changing only the coefficients, which we already know are not considered.

The best-case scenario, however, has improved. If we send in a sorted list, the new algorithm iterates through it once, doing $(n-1)$ comparisons. After that, no elements are swapped, and the algorithm terminates, such that the best case costs $\mathcal{O}(n)$.

# Etter forelesningen

- Vurderingsrubrikken for prosjekt 2 er nå publisert
- Sjekk også oppgaveteksten for oppdateringer