

TDT4102

Prosedyre- og objektorientert programmering, Øvingsforelesning veke 9, 1. mars

The image features a large, stylized blue 'C++' logo. Behind the logo is a snippet of C++ code in a monospaced font, tilted at an angle. The code is:

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

Sivert Krøvel
sivertkr@stud.ntnu.no

Agenda

- Dynamisk minnehandtering
- Deep copy vs shallow copy
- Kopikonstruktør,
tilordningsoperator og
destruktør
- Introduksjon øving 7

Kva betyr «allokere»

Allokere \approx reservere eller «sette av»

- Når vi lagar ein ny variabel, må det *allokerast* plass i minnet til denne variabelen
- Det fins ulike måtar å allokere på, fram til no har vi stort sett lete det skje automatisk

Dynamisk minnehandtering

- Ein «vanlig» variabeldeklarasjon (t.d `int a;`) brukar automatisk allokering
- Variablane har då levetid ut *scopet* dei blei deklarererte i
- Desse vert plassert i ein del av minnet vi kallar *stack*

Dynamisk minnehandtering

- Dynamisk allokerede variabler kontrollerer vi i større grad sjølve
- Dynamisk allokerede variabler plasseres på en egen plass i minnet, ofte kalt heap
- Storleiken på områda vi allokere her kan bestemmas i kjøretid, og levetida til desse er til vi deallokerer dei

Dynamisk minnehåndtering

Illustrerende figur

```
void allocateInt(int* ptr) {  
    ptr = new int;  
}
```

```
int main() {  
    int* myPtr;  
    allocateInt(myPtr);  
    *myPtr = 5;  
}
```

Syntaks

Allokering vha **new**, som returnerer ein **peikar** til minneområdet som er allokert (skjermbilete)

```
int* myInt = new int;
```

Syntaks

Deallokering vha **delete**. Det er god praksis å setje peikaren til **nullptr** etter deallokering (dette skjer ikkje automatisk!)

```
delete myInt;  
myInt = nullptr;
```

Merk: **delete** betyr *ikkje* at peikaren blir sletta, berre at du frigir minnet den peikar til

Minnelekkasje

Dersom vi gløymmer å deallokere (**delete**), vil programmet vårt gradvis «ete opp» minnet vårt. Det dynamisk allokerete minnet vil framleis vere utilgjengelig etter programmet er ferdig. Dette kallar vi minnelekkasje.

Motivasjon

Vi kan opprette objekt i ein funksjon som framleis eksisterer etter at funksjonen er ferdig. Dette gjer vi ikkje i øving 7

Opprettar variabelen her

```
MyArray* generateArray() {
    int size, fill;
    cout << "What should the size be? ";
    cin >> size;
    cout << "What should the array be filled with? ";
    cin >> fill;
    return new MyArray(size, fill);
}
```

Den eksisterer framleis her

```
int main() {
    MyArray* newArray = generateArray();
    delete newArray;
}
```

Motivasjon

Dynamisk allokerete tabellar kan ha variabel storleik. Dette benyttar vi oss av i øving 7. Storleiken kan bestemmast medan programmet køyrer. Det gjer at vi kan lage tabellar som kan endre storleik etter behov (slik som `std::vector` kan, for å nevne eit eksempel)

Dynamisk allokerete tabellar

I øving 3 måtte vi bestemme storleiken på tabellane våre *før programmet køyrer*. I mange tilfelle er det praktisk å kunne bestemme storleiken undervegs, etter behov. Då er dynamisk allokering vegen å gå

Syntaks

Legg merke til at lengda ikkje lenger treng å vere ein konstant

```
int lengd;  
cout << "Skriv inn lengd: ";  
cin >> lengd;  
int* tabell = new int[lengd];
```

Syntaks

Etter du har allokert tabellen, kan han brukast på same måte som dei tabellane vi er vande med

```
tabell[4] = 5;
for (int i = 0; i < lengd; i++) {
    tabell[i] = i*i;
}
```

Syntaks

Dynamisk allokerete tabellar må deallokerast med **delete[]**

```
delete[] tabell;
```

Huskeliste

- Ta vare på peikarane
- Dynamisk allokert minne må alltid deallokerast for å unngå minnelekkasje
- Kvart kall til **new** treng eit kall til **delete**
- **new ...[]** krev **delete[]** (allokerer og deallokerer eit større samanhengande minnområde)
- Set peikarane til **nullptr** etter deallokering for å markere at dei er deallokerte

Agenda

- Dynamisk minnehandtering
- Deep copy vs shallow copy
- Kopikonstruktør,
tilordningsoperator og
destruktør
- Introduksjon øving 7

Klasser med dynamiske variabler

```
class MyArray {
public:
    int length;
    int* data;

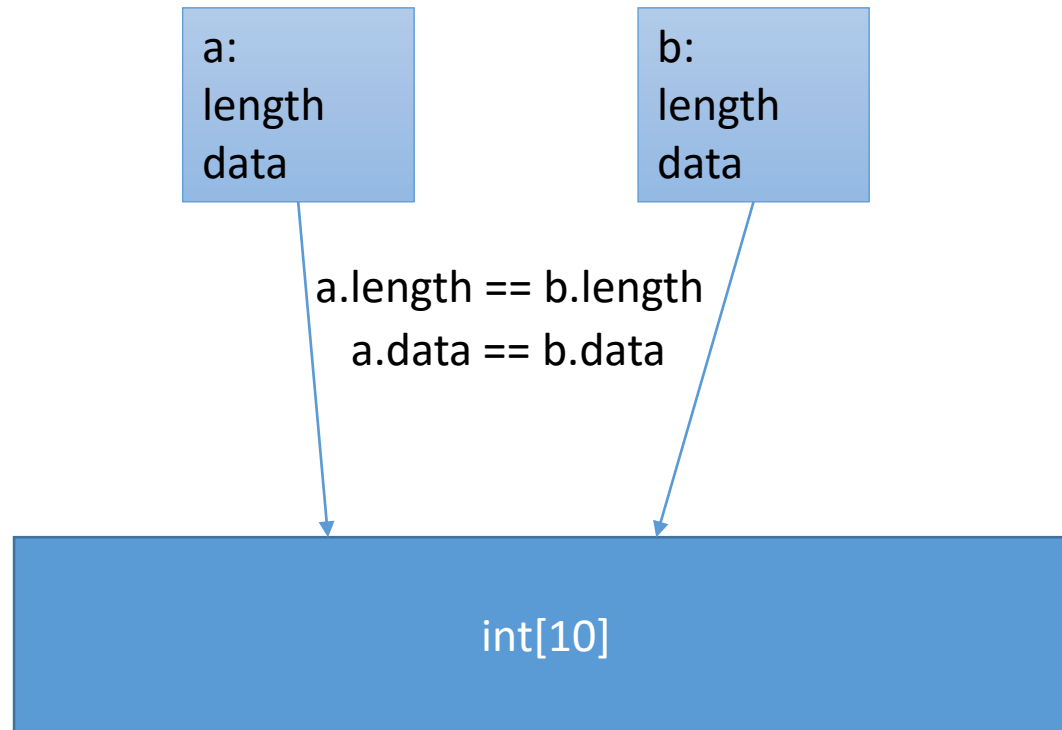
    MyArray();
    MyArray(int size, int fill = 0);
    ~MyArray();
};
```

Kva skjer her?

```
MyArray a(10);  
MyArray b;  
b = a;
```

Først opprettar vi ein MyArray a med 10 plassar.
Deretter opprettar vi ein ny MyArray b, som skal vere
ein kopi av a

Shallow copy ($b = a$)



Begge objekta peikar til *det same* minneområdet!

Shallow copy

- Kompilatorenn lagar automatisk ein kopikonstruktør og tilordningsoperator. Desse implementerer shallow copy
- Shallow copy betyr at kopien deler det same minneområdet med originalen

Problem 1

Kva skjer når to instansar deler på det same minneområdet, og éin av dei endrar på innhaldet?

Problem 1

Kva skjer når to instansar deler på det same minneområdet, og éin av dei endrar på innhaldet?

Innhaldet vert endra for begge objekta!

Problem 2

Kva skjer når to instansar deler på det same minneområdet, og éin av instansane vert destruert?

```
MyArray a(10);  
{  
    MyArray b = a;  
} //b blir destruert her  
a.data[4] = 5;
```


Problem 2

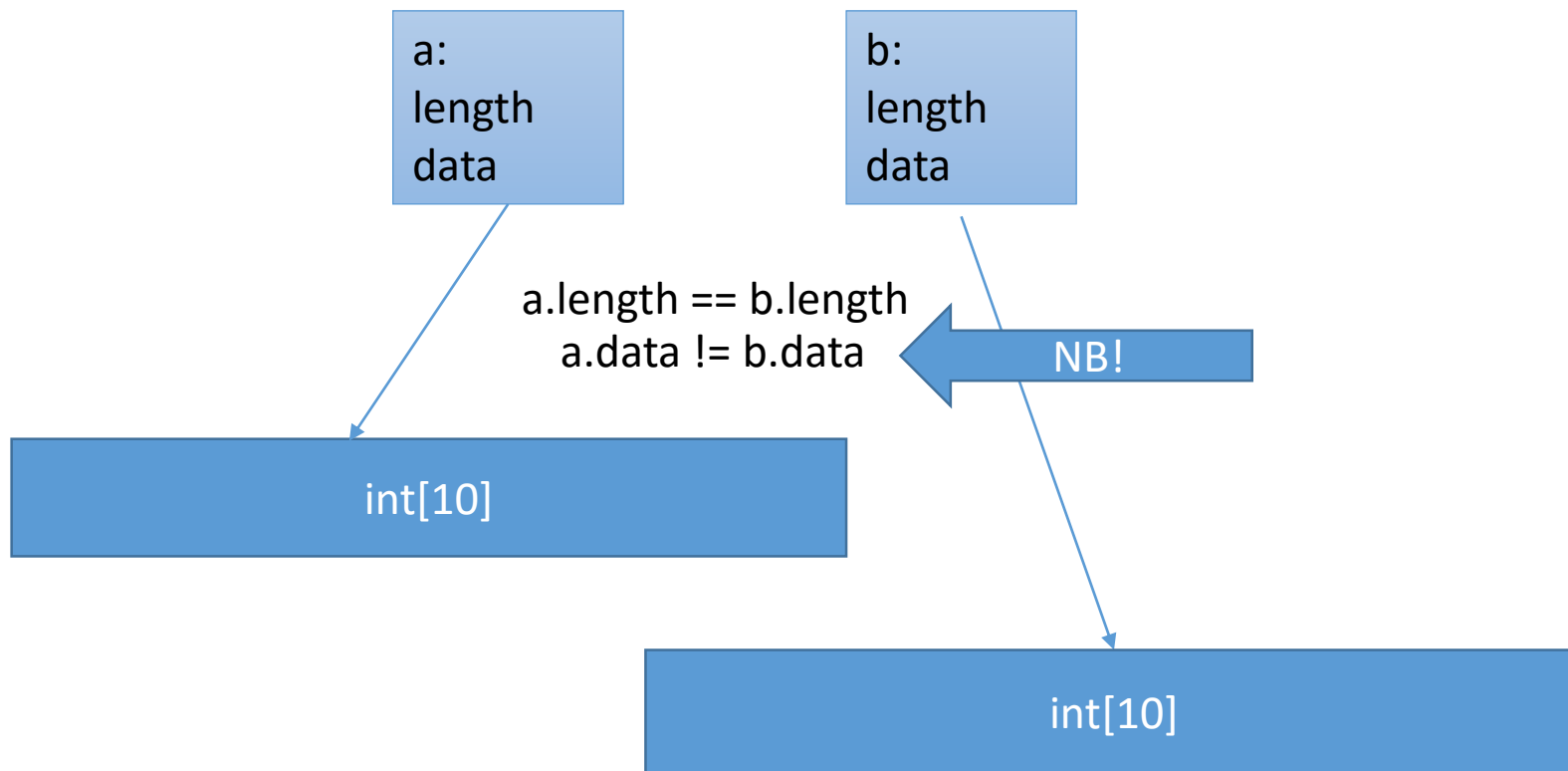
Kva skjer når to instansar deler på det same minneområdet, og éin av instansane vert destruert?

Minneområdet blir frigitt, og utilgjengelig for det objektet som framleis er «i live»

Løysing: deep copy

- Løysinga på desse problema er å allokere eit nytt minneområde til kopien, og putte inn *verdien* frå originalen inn i dette
- Dette kallar vi deep copy
- Implementerast i *kopikonstruktøren* og *tilordningsoperatoren* (operator=)

Deep copy (b = a)



Begge objekta peikar til *ulike* minneområde, men med det same *innhaldet*!

Deep vs shallow copy

Deep copy

- Allokerer nytt minne for det nye objektet
- Kopierer *innhaldet* frå originalen til det nye objektet
- Dei to objekta er uavhengige av kvarandre

Shallow copy

- Kopierer berre *peikaren* til minnet
- Kopien og originalen deler *det same* minneområdet (avhengigheit)
- Brukar mindre minne enn deep copy

Framgangsmåte

- 1) Frigi gammalt allokert minne (om nødvendig)
- 2) **Alloker nytt minne til kopien, like stort som i originalen**
- 3) Kopier innhaldet i originalen over i kopien (for-løkke dersom vi har ein tabell)

Demo

Deep vs shallow copy, intermezzo-oppgåva
(blir ikkje lagt ut, sjekk LF for øving 7 neste veke)

Agenda

- Dynamisk minnehandtering
- Deep copy vs shallow copy
- Kopikonstruktør,
tilordningsoperator og
destruktør
- Introduksjon øving 7

Kopikonstruktøren

```
MyClass(const MyClass& original);
```

- Lagar eit nytt objekt med utgangspunkt i det gamle
- Vert oppretta automatisk dersom vi ikkje inkluderer den i klassedefinisjonen
- Den automatiske implementasjonen brukar shallow-copy

Kopikonstruktøren

Må overlagraast dersom vi brukar *dynamisk* allokering av minne **og** ønsker å lage kopiar som er uavhengige av kvarandre etter dei er oppretta

Skal implementere deep copy

Tilordningsoperatoren

`MyClass& operator=(const MyClass& original);`
 eller `MyClass& operator=(MyClass original);`

- Same formål som kopikonstruktøren, kopierer innhaldet frå høgre operand
- Vert også oppretta automatisk dersom vi ikkje sjølv puttar den i klassedefinisjonen (shallow copy)
- Skal implementere deep copy

Copy-and-swap


Vi ønsker å forhindre duplisering av kode, og det er derfor praktisk å gjenbruke kopikonstruktøren i tilordningsoperatoren, siden dei skal gjøre det same

Copy-and-swap



Call-by-value gjer at kopikonstruktøren kjører, og lagar ein kopi av objektet på høgre side

```
MyArray & MyArray::operator=(MyArray other)
{
    std::swap(this->data, other.data);
    this->length = other.length;
}
```



Her byttar vi om på peikarane

Kopien other, som no har ein peikar til det gamle minnet til venstre operand, vert destruert når funksjonen er ferdig, og minne blir frigitt (destruktøren vert kalla når objektet går ut av scope)

Copy-and-swap forklaring

Denne metoden tek inn høgre operand *by value*, og lagar dermed ein kopi av denne vha kopikonstruktøren. Alt som gjenstår då er å *bytte peikarar*.

Destruktøren tek seg av å deallokere minnet som kopien har tatt over

Copy-and-swap illustrasjon

Illustrasjon

Destruktøren

- Repetisjon: destruktøren skal «rydde opp» etter objektet
- Dette inkluderer å deallokere dynamisk allokert minne
- Hindrar minnelekkasjer
- Hugs å sjekke verdien til peikaren, du kan ikkje deallokere ein **nullptr**!

Destruktøren

```
MyArray::~~MyArray()  
{  
    if (data != nullptr) {  
        delete[] data;  
    }  
}
```


Oppsummering

- Kopikonstruktøren, tilordningsoperatoren og destruktøren høyrer saman. Dersom du treng ein av dei, treng du alle
- Dei to førstnevnte implementerer deep copy slik at kopiar vert uavhengige av kvarandre
- Destruktøren deallokerer dynamisk minne

Agenda

- Dynamisk minnehandtering
- Deep copy vs shallow copy
- Kopikonstruktør,
tilordningsoperator og
destruktør
- Introduksjon øving 7

Øving 7

Handlar om:

- Dynamisk minnehandtering
- Herunder kopikonstruktør, tilordningsoperator og destruktør
- Deep vs shallow copy
- Litt meir operatoroverlagring

Matrix-klassa

- Modellerer ei $M \times N$ matrise, storleiken skal kunne bestemast medan programmet køyrer
- Treng medlemsvariablar til å halde orden på storleiken (to dimensjonar) samt ein tabell til å lagre sjølve innhaldet i matrisa

Matrix-klassa

Kva treng vi av medlemsvariablar?

Matrix-klassa

Kva treng vi av medlemsvariablar?

Dimensjonane lagrast i to heiltalsvariablar

Innhaldet i matrisa kan lagrast i ein ein- eller todimensjonal tabell. Tabellen må allokerast dynamisk, sidan storleiken er ukjend

Lagre data i eindimensjonal tabell

- Lagrast i ein double-peikar (double*)
- Vi treng ein måte å rekne om frå to til éin indeks
- $\text{index} = i * (\text{rows } \textit{eller} \text{ cols}) + j$
- Implementer denne omrekninga i get/set funksjonane og bruk dei

Lagre data i eindimensjonal tabell

- M x N element
- Kolonnene (eller radene) ligg etter kvarandre
- Figur som viser 2x3 matrise:

kolonne 1		kolonne 2		kolonne 3	
1	2	3	4	5	6

1	3	5
2	4	6

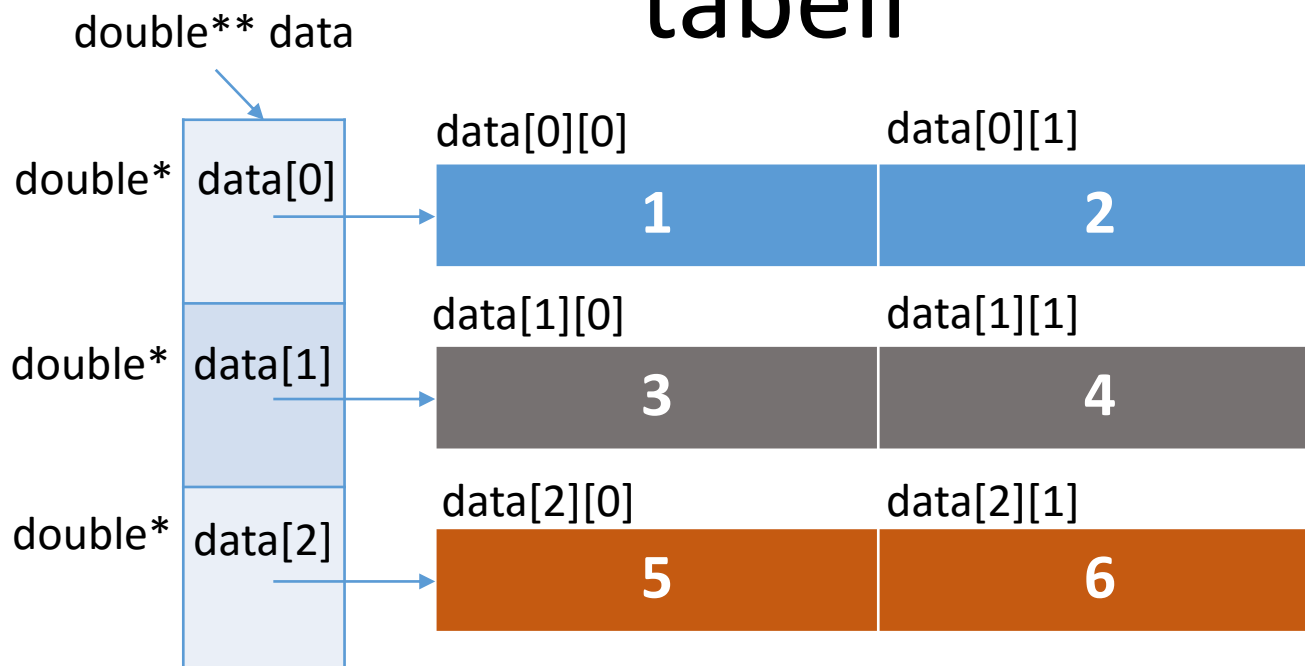
Lagre data i eindimensjonal tabell

- + Enkel å allokere og deallokere
- Meir komplisert å aksessere med to-dimensjonale indeksar

Lagre data i todimensjonal tabell

- Ein tabell med tabellar
- Kvart element i den første tabellen, inneheld ein ny tabell som representerer ei rad eller ei kolonne
- Vi treng ein dobbeltpeikar (ein peikar til ein peikar) `double**`

Lagre data i todimensjonal tabell



1 3 5
2 4 6

Lagre data i todimensjonal tabell

- + Enkel å indeksere med to indeksar
- Meir komplisert å allokere og deallokere

Ein- eller todimensjonal?

Det velger du sjølv. Det er fint å kjenne til begge metoder. Den todimensjonale gir kanskje litt betre forståelse for korleis peikarar og dynamisk minnehandtering fungerer, medan den eindimensjonale løysinga krev «triksing» med indeksar

Intermezzo-oppgåva

Demonstrerer deep vs shallow copy og korleis kopikonstruktøren og tilordningsoperatoren fungerer

Prøv deg fram og sjå kva som skjer. Kjør koden etter kvar endring, og prøv å legge merke til kva som er annleis