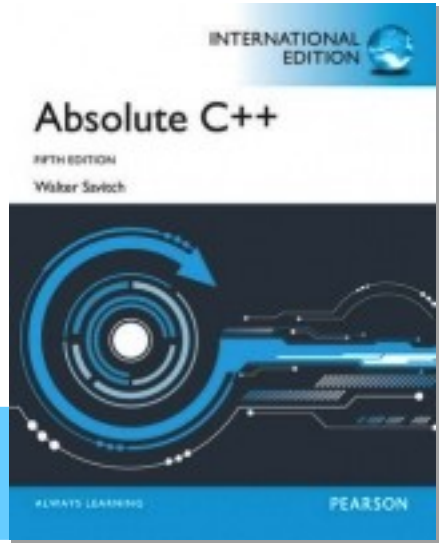
TDT4102
Prosedyre- og objektorientert programmering

Dynamisk allokerte variabler

Innhold

- Automatiske og dynamiske variabler
- Pekere (repetisjon)
- Dynamisk minne: new og delete
- Klasser med dynamisk minne
- Destruktøren
- Kopi-konstruktøren
- Tilordningsoperatoren =

Kap. 10



Automatiske variabler

- "Vanlige" variabler
- Automatisk minnehåndtering
- Scope bestemmer levetid

Automatiske variabler

- main() trenger minne til x, y og z
- average() trenger minne til a, b, sum, result
- Når average() avslutter, frigjøres a, b, sum, resultat
- Minnehåndteringen skjer *automatisk*
- Forutsigbart hvor mye minne main() og average() bruker

```
int main()
{
    double x = 10.0;
    double y = 14.0;
    double z = average(x, y);
    cout << z << endl;
}
```

```
double average(double a, double b)
{
    double sum = a + b;
    double result = sum / 2;
    return result;
}
```

Automatiske variabler

Begrensninger:

- *Minnebruk må være kjent på forhånd*
- Må kjenne til antall variabler på forhånd
- Kun lov med arrays av fast størrelse

Dynamiske variabler

- Variabler der **minnebruk er dynamisk**
- Størrelse kan **bestemmes ved kjøring**
- Dynamiske variabler overlever scope
  - Må selv bestemme når de skal opprettes
  - Må selv bestemme når de skal frigjøres
- Trenger pekere!

Dynamiske variabler

Når trenger vi dem?

a) Arrays der størrelse bestemmes under kjøring

b) Dynamiske datastrukturer
   - f.eks. lenkede lister og trær

c) Store datastrukturer

Repetisjon: Pekere

- Pekere inneholder minneadresser
- Deklarere: type *p;

```
int *p;
```

- Tilordne: p = &var;

```
int a = 10;
p = &a;
```

- Dereferere: *p
  - «Hent det som ligger på minneaddressen som er lagret i pekeren»

```
// lese verdien som p peker til
cout << *p << endl;

// endre verdien som p peker til
*p = 42;
```

Repetisjon: Referanser

- Alias for eksisterende variabler

```
int &r = a;
```

- Pekere og referanser ligner på hverandre
  - En referanse er som en konstant peker (int * const p)
  - En referanse er ”låst” til en minneadresse

Bruk av pekere

- Når vi ønsker å endre variabler utenfor funksjonens scope
- Navnet til en array er en peker til første element
  - *kan bruke pekere og pekeraritmetikk på arrays*

```
void mySwap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

int main(){
    int a = 2;
    int b= 4;
    mySwap(&a, &b);
}
```

```
int arr[10] = {};

for (int *p = arr; p < &arr[10]; p++){
        cout << *p << endl;
}
```

new

- Allokere minne dynamisk
- Returnerer en *peker* til minnet
- Konstruktør-syntaks for å initialisere variabler

```
// Deklarering, allokering og tilordning i samme linje
double *p1 = new double(5.0);

// Deklarering først, allokering og tilordning etterpå
double *p2;
p2 = new double(10.0);

// Deklarering og initialisering
double *p3(new double(15.0));
```

delete

- Frigjør dynamisk minne allokert med new
- Må selv frigjøre minne vi ikke lenger har bruk for!
- Etter delete er innholdet i pekeren *ugyldig*
  - Må ikke røre frigjort minne!
  - Frigjort minne vil bli gjenbrukt

Hvis vi glemmer delete...

- Alt minne frigjøres når programmet avslutter...
- Minnelekkasje:

```
while (true) {
    double *d = new double(1.0);
    kalkuler(*d);
    delete d;
}
```

Når en peker ikke peker...

- Rett etter deklarering har pekere en udefinert verdi
- Kan bruke NULL eller nullptr (C++11)
  - Pekeren "peker ikke til noe"
- God praksis å sette peker til NULL etter delete

```
int *p = nullptr; // eller NULL

if (p == nullptr) {
   cout << "p peker ikke til noe..." << endl;
} else {
   cout << "p peker til " << *p << endl;
}
```

new []

- Allokerer en *dynamisk array*
- Kan bestemme størrelse under kjøring
- Returnerer en *peker* til første element

```
int size = 0;
cout << "Hvor mange tall? ";
cin >> size;
int *arr = new int[size];
```

delete []

- Frigjør en dynamisk array
- delete [] vet hvor stort array er :-)

```
delete [] arr;
arr = nullptr;
```

Flerdimensjonale dynamiske array

- Dynamiske arrays er kun 1-D
- Måter å lage flere dimensjoner:

(a) Bruke peker til peker

(b) Regne om fra N-D indeks til 1-D

2-D array: peker til peker

1. Deklarer array som peker til peker **
2. Alloker array av pekere til radene
3. Alloker en array for hver rad

```
int **a = new int*[ROWS];

for (int i = 0; i < ROWS; i++) {
   a[i] = new int[COLUMNS];
}
```

```
for (int i = 0; i < ROWS; i++) {
   delete [] a[i];
}
delete [] a;
```

✓ *Todimensjonal indeksering [i][j]*

```
a[i][j] = 42;
```

✘ *Omfattende allokering og frigjøring*

✘ *Minne ikke sammenhengende*

1-D array som 2-D array

1. Deklarer array som peker *
2. Alloker array med størrelse rows * columns
3. Regn om fra 2-D indeks til 1-D

```
int *a = new int[ROWS * COLUMNS];
```

```
delete [] a;
```

✓ *Enkel allokering og frigjøring*

✘ *Tungvint addressering*

```
a[i * ROWS + j] = 42;
```

Klasser og dynamisk minne

- Objekter kan inneholde dynamiske variabler
- Allokert med new i konstruktøren

- *Hva skjer når objekter med dynamisk minne går ut av scope?*

```
class Array {
private:
    int size;
    int *arr;
public:
    Array(int size) {
        this->size = size;
        arr = new int[size];
    }
};
```

Destruktøren

- Må kalle delete på dynamiske medlemsvariabler før selve objektet frigjøres
- Destruktør: gjør det "motsatte" av konstruktør
- Default-versjonen sletter kun vanlige medlemsvariabler
- Deklareres som konstruktører, men med prefikset ~

```
Array::~Array(){
    delete [] arr;
}
```

Destruktøren

- Kalles automatisk når:
  - delete kalles på dynamiske variabler

```
Array *a = new Array(400);
delete a; // her kalles destruktøren på a
```

  - automatiske variabler går ut av scope

```
{
   Array b(200);
} // her kalles destruktøren på b
```

Kopiering og tilordning

- Hva skjer med dynamiske medlemsvariabler når vi:
  - Kopierer?

```
Array a(100);
Array b(a);  // kopiering
```

  - Tilordner?

```
Array c = a; // tilordning
```

Shallow copy

- Default for tilordning og kopiering
- Tilordne/kopiere medlem for medlem
- Alle medlemsvariabler i **b** bli tilordnet tilsvarende medlemsvariabel i **a**
- Bare "overflaten" som kopieres
- Kopiering av pekere:
  - Mister adressen som opprinnelig ble pekt til: minnelekasje
  - Får to objekter med pekere til samme allokerte minne!

```
Array b(a);
Array b = a;
```

```
b.size = a.size;
b.arr = a.arr;
```

Deep copy

- Lage fullstendig kopi
  - Kopi av medlemsvariabler
  - Kopi av dynamiske allokerte variabler
- Ønsker tilordning og riktig minnehåndtering
  - Opprydding av gamle dynamiske variabler
- Må implementere egen **kopi-konstruktør**
- Må overlagre **tilordningsoperator** =

Kopi-konstruktøren

- Konstruktør med typen selv som eneste parameter

```
Type::Type(const Type& other);
```

- Kalles automatisk: call-by-value, return-by-value
- Default kopi-konstruktør: shallow copy

Kopi-konstruktøren

- Deep copy: trenger egen kopi-konstruktør

```
Array::Array(const Array& other) {
    size = other.size;
    arr = new int[size];
    for (int i = 0; i < size; i++) {
        arr[i] = other.arr[i];
    }
}
```

Tilordningsoperatoren =

- Default operator: shallow copy
- Deep copy: må overlagre operator=
- Vanlig teknikk: copy-and-swap

Copy-and-swap

```
Array& Array::operator=(Array rhs) {
    std::swap(size, rhs.size);
    std::swap(arr, rhs.arr);
    return *this;
}
```

- Tar inn kopi som **`rhs`** (call by value)
- Swap: **`this`** får ny kopi og **`rhs`** får gammel data
- Destruktor kalles automatisk for **`rhs`** (scope)
  - Frigjør gammel data!

Forhindre kopiering og tilordning?

- Slette default kopikonstruktør og operator=
- `=delete` etter deklarasjonen

```
class Array {
    ...
    Array(const Array& other) = delete;
    Array& operator= (Array rhs) = delete;
    ...
};
```

The rule of three…

1. Destruktør
2. Kopikonstruktør
3. Tilordningsoperator

Hvis vi lager én så trenger vi vanligvis alle!

Rekursive datatyper

- Klasser kan ha medlemsvariabler av samme type som seg selv
- Men bare hvis de er pekere!
- Ellers ville vi endt opp med en ”uendelig” datastruktur!

```
class Person {
private:
    string name;
    Person *father;
    Person *mother;
    …
};
```

Forward deklarering

- Må bruke et lite ”triks”:
  Forward deklarering
- Ber kompilatoren om å akseptere Person som type før den er deklarert

```
class Person;

class Person {
private:
    string name;
    Person *father;
    Person *mother;
    …
};
```

Når bruker vi pekere?

- Når vi trenger "alias" for samme variabel
- Dynamisk minne: `new`, `delete`, `new[]`, `delete[]`
- Rekursive datatyper
- Store datastrukturer

Oppsummering

- Automatiske og dynamiske variabler
- Pekere (repetisjon)
- Dynamisk minne: new, delete, new[] og delete[]
- Klasser med dynamisk minne
- Destruktøren
- Kopi-konstruktøren
- Tilordningsoperatoren =