30
Objekt-orienteret Programmering i Andre Sprog.

Abstrakte datatyper i Pascal.

Abstrakte datatyper i Modula og Ada.

C++

- Overordnet organisering
- Instantiering og initialisering.
- Interfaces.
- Nedarvning.

PS1 -- OOP i andre sprog

Noter

Temaet i dette kapitel er, hvordan man kan praktisere objekt-orienteret programmering i andre sprog end Eiffel.

Først ser vi kort på hvordan man med større eller mindre disciplin drage fordel af objekt-orienterede principper i ikke OO sprog.

Dernæst vender vi os mod C++, som *er* et objekt-orienteret sprog, og nok også det mest udbredte.

Simulering af abstrakte datatyper I Pascal.

```
...
type bankkonto = record
                   kunde: Person;
                   balance, rentesats: real;
                 end

(* andre typer *)

procedure opret_konto(var k: bankkonto);
begin ... end

function balance(var k: bankkonto): real
begin ... end

procedure add(var k: bankkonto); (* internal *)
begin ... end

procedure hæv(var k: bankkonto, b: real)
begin ... end

procedure indsæt(var k: bankkonto, b: real)
begin ... end
```

- Tilgang til data via procedurer eller funktioner.
- Definere operationsinterface til data.
- Opnå repræsentationsuafhængighed i anvendelsen af data.
- Udelukkende et resultat af *programmørens disciplin*.
- Vanskeligt/umuligt at gruppere sammenhængende definitioner hensigtsmæssigt.

Noter

I sprog ala Pascal kan man have meget glæde af at programmere abstrakte datatyper. Men der er er næsten intet i sproget der understøtter dette, ud over procedurer, funktioner, og brugerdefinerede typer. De øvrige aspekter af abstrakte datatyper skal frembringes ved programmør-disciplin, og flittig brug af kommentarer. Eksempelvis kan operationsinterfacet af en ADT kun adskilles fra de interne operationer ved brug af kommentarer.

De mere avancerede aspekter i objekt-orienteret programmering, såsom nedarvning, generiske typer, polymorfi og dynamisk binding, er vanskelige at håndtere og opnå. Vi har tidligere set, at variant-records til en hvis grad kan gøre det ud for ét niveau af specialisering.

Moralen er, at dataabstraktion er en *ide*, som kan praktiseres i de fleste sprog.

Abstrakte datatyper
I sprog med et modul-begreb.

Modula/Ada agtige sprog.

```
Module Bank
export bankkonto, opret-konto, balance, hæv,
        indsæt

 type bankkonto = record
                    kunde: Person;
                    balance, rentesats: real;
                  end

 procedure opret_konto(var k: bankkonto);
 begin ... end

 function balance(var k: bankkonto): real
 begin ... end

 procedure hæv(var k: bankkonto, b: real)
 begin ... end

 procedure indsæt(var k: bankkonto, b: real)
 begin ... end

 procedure add(var k: bankkonto, b: real)
 begin ... end
End Bank
```

- Definere operationsinterface til data.
- Opnå repræsentationsuafhængighed i anvendelsen af data.
- Indkapsling og *sprog-påtvunget beskyttelse* overfor udisciplineret tilgang til data.
- Separat kompilering af hver abstrakt datatype.
- Vanskeligt at opnå en nedarvnings-effekt

Noter

Eksemplet på denne slide er ikke forsøgt gjort præcis, og er hverken i Modula eller Ada syntaks.

Det skal bemærkes, at et modul ikke kan instantieres som et objekt. Modulet, som har et navn, eksporterer en type af et andet navn. Denne type kan anvendes til definition af variable, og dermed statisk eller dynamisk instantiering af bankkonti.

Et modul er således blot en syntaktisk ramme omkring nogle definitioner. Rammen tillader dog kontrol over synlighed af modulets definitioner, således at noget kan holdes skjult internt i modulet.

Både Modula og Ada understøtter programmøren i at definere klare interfaces af eksterne operationer. Dette gøres i separate interface moduler. Detaljerne af operationerne, samt interne operationer gives i tilhørende definitions-moduler. Modula og Ada er således forskellige fra Eiffel, som jo tillader os at udtrække en inteface definition med ***short*** efter at hele klassen er skrevet.

C++

- Et objekt-orienteret sprog, som (stort set) indeholder C som en delmængde.
- Et systemprogrammeringssprog.
- Orienteret mod høj grad af effektivitet.
- Deallokering af dynamisk allokerede data skal ske eksplicit.
  Ingen automatisk garbage collection.
- Antageligt det mest udbredte objekt-orienterede sprog.
- Designet af en dansker, Bjarne Stoustrup, hos AT&T (Bell Labs).

Noter

I resten af dette kapitel vil vi give en kort oversigt over vigtige og relevante dele af C++.

C++ : Overordnet organisering.

Scope.

- Muligt at have "globale" definitioner, som er synlige
  - i et helt program.
  - i hele filen, hvori definitionen befinder sig.

  Der kan dog være huller i scopet.

Praktisk behændigt.
Nedsætter genbrugsværdi

Fil-organisering.

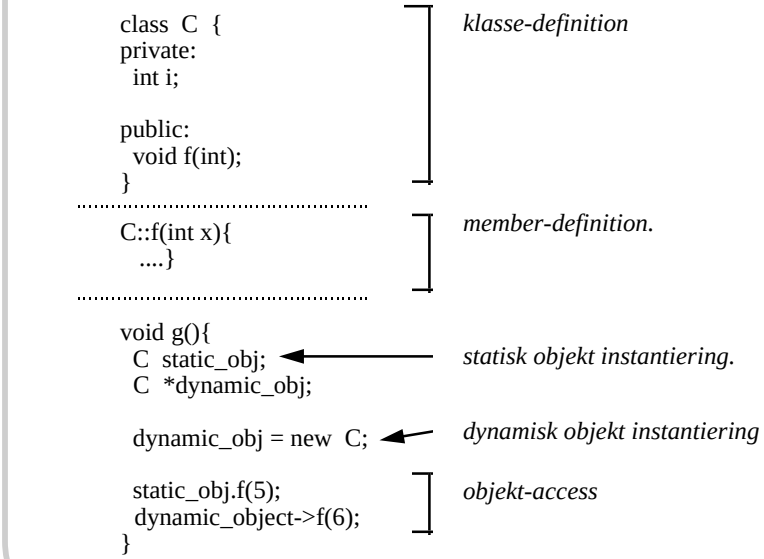
- Ingen tvungen organisering ala ' én klasse pr. fil'.
- Mulig organisering af et objekt-orienteret C++ program:
  - .h fil med klasse-definition,
    dog uden defintion af kroppene af klassens operationer.
  - .c fil med fuld definition af klassens operationer.
- "Hovedprogram": funktion med navn *main.*
- C++ omgivelser understøtter traditionelt ikke automatisk, minimal re-compilering.

Noter

I Eiffel er klassen den alt-dominerende abstraktionsform, og enhver procedure/funktion er indeholdt i en klasse.

Dette er ikke tilfældet i C++. Her kan man lave sædvanlig programmering, ala det man altid har gjort i C. Klasserne er et supplerende tilbud, som man kan bruge eller lade være.

C++ : Klasser, operationer, instantiering, og object-access.



PS1 -- OOP i andre sprog

Noter

Den øverste del er det typiske indhold i .h filen. Den midterste del udgør typisk .c filen.

C programmører vil nikke genkendende til syntaksen

C static_obj

Dette svarer til

static_obj: C

i Eiffel og Pascal.

Bemærk de to forskellige former for objekt tilgang, dels med dot (.), og dels med pil (->).

C++ : Eksempel på en klassedefinition.

```
class bank_account{

private:
  float ballance;
  float interest_rate;

public:
  bank_account(float amount, float interest);
  bank_account();

  ~bank_account();

  void deposit(float);
  void withdraw(float);
  float amount_on_account();
 };
```


Noter

Eksemplet viser interface definitionerne i en typisk .h fil.

C++ : Eksempler på definition af operationer i en klasse.

```
bank_account::bank_account(float init, float ir) {
  ballance = init; interest_rate = ir;
}

bank_account::bank_account() {ballance = 0; interest_rate = 0;
}

bank_account::~bank_account() {};

void bank_account::deposit (float amount) {
  ballance += amount;
}

void bank_account::withdraw (float amount) {
  ballance -= amount;
}

float bank_account::amount_on_account() {
  return ballance;
}
```


Noter

I forlængelse af forrige slide, vises her de simple funktionsdefinitioner, som typisk gives i en .c fil.

C++ : Anvendelse af en klasse.

```
int main() {
  bank_account b1(500,1.0), b2(700, 2.0);
  bank_account *bp = new bank_account(800, 3.0);

  b1.deposit(100);
  b1.withdraw(50);

  b2.deposit(300);

  bp->withdraw(100);

  cout << "b1 amount " << b1.amount_on_account() << "\n";
  cout << "b2 amount " << b2.amount_on_account() << "\n";
  cout << "bp amount " << bp->amount_on_account() << "\n";
}
```


Noter

Endelig viser denne slide en mulig anvendelse af bank_account klassen. Igen viser vi både en statisk og en dynamisk allokeret bankkonto.

De tre nederste linier er blot C++'s måde at lave output på.

C++ : Instantiering af klasser og initialisering af objekter.

Statisk instantiering:

C static_object

Dynamsik instantiering:

C *dynamic_object

Statisk instantiering og initialisering

C static_object(*constructor-parametre)*

Dynamsik instantiering og initialisering

C *dynamic_object = C(*constructor-parametre)*

Constructor:

- Operation, af sammen navn som klassen, hvis formål det er at initialisere klassens data.
- C++ tillader samlet definition af variable, dynamisk instantiering samt initialisering.
- Constructorer er ofte 'overloadede'.

Destructor:

- Beregnet til oprydning inden deallokering af et objekt.

PS1 -- OOP i andre sprog

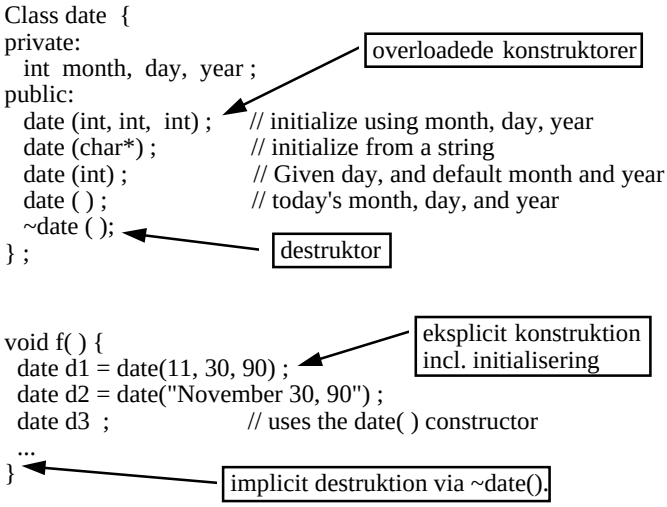
Noter

Denne slide introducerer konstruktorer i C++, og deres mulige anvendelse.

Konstruktorer modsvarer Eiffels create, men konstruktorerne er langt mere fleksible at have med at gøre.

Næste slide viser nogle praktiske eksempler på definition og anvendelse af konstruktorer.

C++ : Eksempel på konstruktorer og destruktorer.



PS1 -- OOP i andre sprog

Noter

På denne slide vises en skitse af klassen date, som har ikke mindre en fire forskellige konstruktor definitioner. Disse adskiller sig i forhold til hinanden ved forskellige signaturer, altså forskellige parametre, på baggrund af hvilke objekter af typen date kan initialiseres.

Nederst vises eksempler på anvendelse af nogle af konstruktorerne.

C++ : Interfaces.

member funktion:
operation i en klasse.

- C++ opdeler operationer og data i tre kategorier:
  - **Private members**:
    Kun synlige i member funktioner af klassen, hvori de er declarerede.

    Relevant for programmøren af klassen.
  - **Protectede members:**
    Synlige i member funktioner af klassen, samt i member funktioner af afledte klasser.

    Relevant for programmøren af klassehierarkiet.
  - **Public members**:
    Synlige overalt.

    Relevant for programmøren, som anvender klassen.
- I klienter af C++ klasser er der en systematisk, syntaktisk forskel på funktionskald (af funktion member uden parametre) og reference af en variabel (data member),

Noter

C++ kategoriserer definitionerne i klassen. Eiffel, derimod, arbejder med eksportlister. De eksporterede navne svarer til C++ public members.

Nyt i forhold til Eiffel er private "features", som ikke kan ses i evt. subklasser.

Public features udgør klient interfacet.

Det nederste punkt er igen en forskel i forhold til Eiffel, som jo gør en dyd ud af ikke at gøre forskel på funktionskald uden parametre og attribut-anvendelse.

C++ : Operationer der tilhører mere end én klasse.
Friends.

Problemet:

Hvis en operation arbejder symmetrisk på data i to eller flere klasser, er det

1. unaturligt at placere operationen i netop én af disse.
2. ineffektivt at tilgå data i nogle af klasserne via operations-interfacet.

En løsning (friends):

- Placere operationen eksternt i forhold til klasserne.
- Bevilge ekstraordinær adgang (*venskab*) til private data.
- Venskab bevilges af klasser, der har noget at skjule.
  Venskab kan ikke fordres af funktioner, der vil "accesse" noget.

Noter

Der er intet i Eiffel som modsvarer friend mekanismen i C++.

C++ : Eksempel på friends.

```
class vector {
 friend vector
     multiply (matrix&,  vector&) ;
private:
  float  v [4] ;
public:
  // vector operations
};
```

```
class matrix {
  friend vector
    multiply (matrix&,  vector&) ;
private:
  vector v [4] ;
public:
 // matrix operations
} ;
```

```
vector multiply (matrix& m,   vector& v)
{ vector r;
   for (int i = 0; .....) { ...
    for (int j = 0; ......) {...
       r.v[i] += m.v[i][j] * v.v[j] ; }}
    return r; }
```

Direkte adgang
til både vector og matrix

PS1 -- OOP i andre sprog

Noter

C++ : Nedarvning.

- To forskellige former for nedarvning

  - offentlig nedarvning (default).
  - privat nedarvning

  Har betydning for klient-interfacet af subklassen.

- Komplikationer vedrørende initialisering af objekter af afledte klasser.

- Dynamisk binding gennem definition af virtuelle funktioner.

- Typekonvertering af objekter.
  Implicit aktivering af system- såvel som brugerdefinerede typekonverteringer.

- Multipel nedarvning:

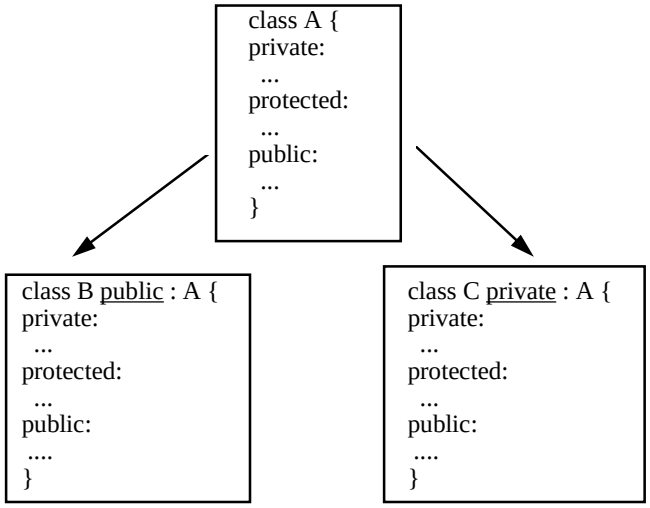
  - Ved navne-sammenfald i to superklasser løses tvetydigheden ved eksplicit klasse-kvalificering af *navne-anvendelser*.

  - Public/privat nedarvning bestemmes individuelt for hver af superklasserne.

  - Kontrol-mekanismer til 'gentagen nedarvning' fra den samme klasse.

Noter

C++ : Synlighedskontrol og offentlig/privat afledning (1).

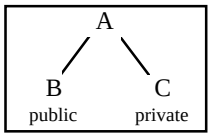


A er en public basis klasse

A er en privat basis klasse

Noter

C++ : Synlighedskontrol og offentlig/privat afledning (2).



	private A members	protectede A members	public A members
public basis klasse	usynlige i B	protectede i B	public i B
privat basis klasse	usynlige i C	private i C	private i C

Noter

Ved en privat nedarvning angiver den specialiserede klasser ene og alene, hvilket interface klassen har til klienter.

C++ : Konstruktorer i forhold til afledte klasser.

```
class   A { ...                 class B : public A { ...
public:                         public:
  A( int ) ;                      B (int, char*) ;
}                               }
```

Problemstillinger:

1. Hvordan initialiseres A-delen af et B-objekt?
   Tvetydighedsproblemer pga. overloadede konstruktorer.
2. I hvilken rækkefølge oprettes/initialiseres A-delen og B-delen af et B-objekt.

ad 1

```
B::B(int  i,  char* str )  : A( i )  {
  ....
}
```

Deklarativ angivelse af, at A-delen initialiseres med konstruktoren A(i).

ad 2

Objekter konstrueres fra basis-klasser mod afledte klasser.

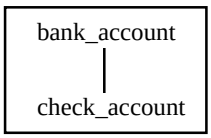
Noter

Det som illustreres på denne slide svarer i Eiffel til, hvordan create i en subklasse kan kalde create i en superklasse (via passende renaming). C++ understøtter en helt særlig syntaks for dette.

C++ : Eksempel på definition og anvendelse af virtuel funktion.

Rentetilskrivning-teknikken er afhængig af bank-konto typen.

```
class bank_account {
protected:
  ...
public:
 virtual float ascribe_interest();
  };
```




```
float  bank_account::ascribe_interest(){
  float interest =
    (ballance * interest_rate) / 100;
  ballance += interest;
  return interest;}
```

```
float  check_account::ascribe_interest(){
  float interest =
    ((ballance - 2000) > 0)?
      ((ballance - 2000) * interest_rate) / 100 :
      0;
  deposit_with_reason(interest, "Interest");
  return interest;
}
```


Noter

I Eiffel opnår vi dynamisk binding ved systematisk redefinition i forbindelse med nedarvning. Mekanismen i C++ er anderledes, idet visse udvalgte operationsdefinitioner kan mærkes som virtuelle. For sådanne funktioners vedkommende er det den dynamiske type af objektet, som afgør hvilken udgave af den virtulle funktion, som bliver kaldt.

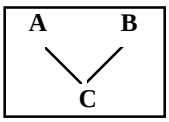
I forelæsningen "Nedarvning I" har vi allerede været inde på virtuelle funktioner.

C++ : Multipel nedarvning og eksplicit navnekvalifikation.

```
class A {
public:
  void f( ) ;
... } ;

class B {
public:
  void f ( );
 ... } ;

class C : public A, public B {
void g( ) {
   f( ) ; // fejl!
   A::f() ;   B::f() ;  // OK.
   ... }
}
```

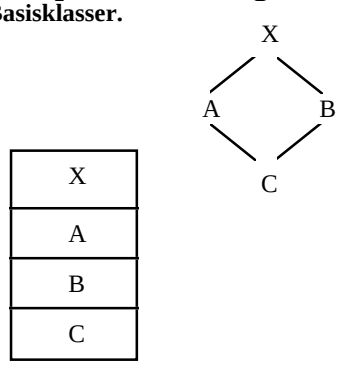


Det er tvetydigt om A::f( ) eller B::f( ) refereres.

Noter

C++ : Multipel nedarvning.

Virtuelle Basisklasser.




```
class X { ... } ;
class A : virtual public X { ... };
class B : virtual public X { ... } ;
class C: public A, public B { ... } ;
```

```
class X { ... } ;
class A : public X { ... };
class B : public X { ... } ;
class C: public A, public B { ... } ;
```

X *er en virtuel basisklasse af* A *og* B.

Noter

Ligesom vi kan annotere en nedarvingsrelation i C++ med public og private, kan vi endvidere annotere denne med virtual. Dermed får vi kontrol over multipliciteten af allokering af superklasser, som kan nåes via mere end én vej i klassehierarkiet.

Eiffel understøtter noget lignende, men på enkeltfeature niveau, og via renaming mekanismen.

Bemærk, at virtual i denne sammenhæng ikke har noget at gøre med virtuelle funktioner, som vi har diskuteret tidligere i kapitlet.