Dagens tema

- Kjøresystemer
  (Ghezzi&Jazayeri 2.6, 2.7)
  — Repetisjon
  — Språk med rekursjon (C3) og blokker (C4)
  — Statisk link
  — Dynamisk allokering (C5)
  — Parameteroverføring

Repetisjon: Statiske språk uten rekursive metoder (C1 og C2)

Minnebehovet vil her være statisk kjent, og kompilatoren kan gi hver variabel en fast plass i minnet.

Vi lar hver metode få sin egen aktiveringsblokk med plass til lokale variable. Vi trenger også plass til returadressen.

Separat kompilering (C2′)

Under kompilering kan en variabel nå ikke tilordnes til en bestemt lokasjon. Vi må istedenfor beregne variablens offset i forhold til starten på den aktuelle aktiveringsblokken. Linkeren vil så samle all informasjonen og tilordne absolutte adresser.

INF3110 – Programmeringsspråk

Repetisjon: Statiske språk uten rekursive metoder (C1 og C2)

Minnebehovet vil her være statisk kjent, og kompilatoren kan gi hver variabel en fast plass i minnet.

Vi lar hver metode få sin egen aktiveringsblokk med plass til lokale variable. Vi trenger også plass til returadressen.

Separat kompilering (C2′)

Under kompilering kan en variabel nå ikke tilordnes til en bestemt lokasjon. Vi må istedenfor beregne variablens offset i forhold til starten på den aktuelle aktiveringsblokken. Linkeren vil så samle all informasjonen og tilordne absolutte adresser.

Forelesning 11 – 5.11.2003 2/25

Repetisjon: Rekursive metoder (C3)

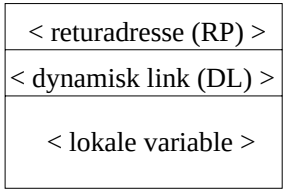
For hver metode trenger vi like mange aktiveringsblokker som antall rekursive kall. Minnebehovet er dermed ikke lenger statisk kjent.

Vi organiserer derfor minnet som en *stakk* av aktiveringsblokker. Vi trenger da to "systemvariable":

- `current`, en peker til gjeldende aktiveringsblokk (ligger i `D[0]`).
- `free`, en peker til første ledige plass på stakken (ligger i `D[1]`).

Repetisjon: Rekursive metoder (C3)

I tillegg har hver aktiveringsblokk en peker, dynamisk link (DL), tilbake til forrige aktiveringsblokk:



Returverdier

Vi lar en eventuell returverdi ligge *mellom* de to aktiveringsblokkene til kalleren og den som blir kalt.

Metodekall

Følgende må gjøres av *kalleren*:

1. Sette av plass til returverdien.
2. Lagre returpekeren.
3. Sette den nye blokkens dynamiske link til gjeldende aktiveringsblokk.
4. Sette `current` til å peke på den nye aktiveringsblokken.
5. Sette `free` til å peke på neste ledige plass.
6. Sette `ip` til å være første instruksjon i metoden som kalles.

Metoderetur

Følgende må gjøres av *metoden som returnerer*:

1. Slette aktiv aktiveringsblokk.
2. La forrige aktiveringsblokk bli aktiv.
3. Hoppe tilbake til kallstedet.

C4: Språk med blokker

To varianter:

- C4′: Tillater lokale variable i en sammensatt setning.
- C4″: Alle deklarasjoner (også av metoder) kan plasseres inne i lokale blokker.

Hvorfor blokkstruktur?

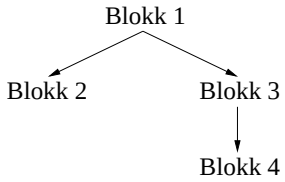
- Kontrollere skopet til variable.
- Definere variables levetid.
- Dele programmet i hensiktsmessige biter.

C4′: Indre lokale deklarasjoner

Her kan vi ha lokale variable i en sammensatt setning:

```
1  void f()                  // Blokk 1
2  {
3    int a, b;
4
5      :
6    if (a < b) {            // Blokk 2
7      int temp = a;
8      a = b;  b = temp;
9    }                       // Slutt blokk 2
10     :
11   while (a > b) {         // Blokk 3
12     int x, y;
13       :
14     if (...) {            // Blokk 4
15       int z;
16         :
17     }                     // Slutt blokk 4
18   }                       // Slutt blokk 3
19 }                         // Slutt blokk 1
```

For å få oversikt over selve blokkstrukturen kan vi lage et "static nesting tree":



Implementasjon

Dette krever minimale utvidelser i forhold til C3. To måter å gjøre det på:

- Statisk sette av plass til *alle* variable i aktiveringsblokken til den omsluttende prosedyren.

  (Figur på tavla.)

- Dynamisk tildeling av minne når man går inn i ny blokk.

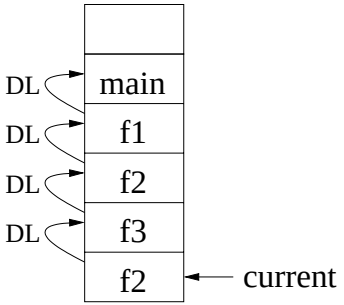
C4″: Metoder inni metoder

Alle deklarasjoner (også av metoder) kan plasseres inne i lokale blokker.

Eksempel **(Hvilke metoder kan kalles hvor?)**

```
1  void f1()
2  {
3    f2()
4    {
5      f3()
6      {
7      }
8    }
9  }
10
11 void main()
12 {
13 }
```


Anta at vi har en kallsekvens som gir opphav til følgende run-time stakk:



Hvilke variable skal dette siste kallet på `f2` kunne få tak i?

Hvordan kan `f2` finne frem til disse variablene?

Statisk link

Hver aktiveringsblokk må også inneholde en peker, statisk link (SL), til aktiveringsblokken for omkringliggende blokk:

< returadresse (RP) >
< dynamisk link (DL) >
< statisk link (SL) >
< lokale variable >

En variabel vil alltid kunne nås i en bestemt "avstand" fra den aktuelle blokken:

- En lokal variabel har avstand = 0.
- Variabel deklarert i direkte omsluttende blokk: avstand = 1.
- ...

Hver referanse til en variabel gjøres om til et par:
⟨ avstand, offset ⟩ under kompileringen.

I SIMPLESEM finner vi verdien til en variabel angitt ved ⟨ avstand, offset ⟩ ved å følge statisk link avstand ganger og legge til offset.

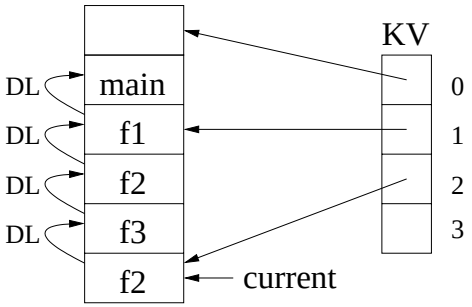
Verdien til en variabel kan da skrives D[fp(avstand) + offset], der fp-funksjonen (for "frame pointer") er definert ved:

$$fp(d) == \textbf{if } d = 0 \textbf{ then } D[0] \textbf{ else } D[fp(d-1)+2]$$

Hvordan sette statisk link?

Kontekstvektor

For å spare tid, har noen systemer en kontekstvektor som peker til de aktiveringsblokkene som for tiden er synlige:



Fordeler og ulemper

+ Variabelaksessen blir raskere.

– Det blir mer jobb ved hvert metodekall og -retur.

Dynamisk allokering

Størrelsen kjent ved aktivering

Eksempel:

```
1 begin
2   integer n;
3   n := inint;
4   begin
5     integer array dynarray(1:n);
6     ...
7   end;
8 end
```

Størrelsen på aktiveringsblokken er først kjent ved aktivering.

Dynamisk allokering

Abstrakt implementasjon:

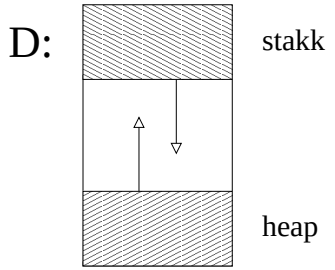
- Ved kompilering:
  - Hver dynamisk array får en deskriptor med plass til peker til starten av arrayen, samt nedre og øvre grenser.
  - All array-aksessering oversettes til å gå via denne deskriptoren.
- Under kjøring:
  - Ved evaluering av en dynamisk array-deklarasjon: øk aktiveringsblokken med nødvendig plass, og oppdater verdiene i deskriptoren.

Full dynamisk allokering

```
1  struct node {
2     int info;
3     node* left;
4     node* right;
5  };
6  ...
7  node* n = new node;
```

Problem: Levetiden til et objekt er generelt *ikke* avhengig av blokken hvor objektet opprettes!

Løsning: Vi setter av et eget område i lageret til denne typen data. Dette området kalles en heap.



Parameteroverføring

- Hvordan overføres parametre?
- Hva slags mekanismer finnes det?
- Hvordan kan disse implementeres?

Vi skal se på følgende eksempel:

```
1  int a = 1;
2
3  void f(int x)
4  {
5    x = x+1;
6  }
7
8  void main()
9  {
10   f(a);
11 }
```

SIMPLESEM:

D

0	< current >
1	< free >
2	< a >

f:

0	< RP >
1	< DL >
2	< SL >
3	< x >

Referanseparametre

Her sendes det over en *referanse* til hvilken lagercelle parameteren ligger i. (Call by reference.)

Overføring av aktuell parameter `a` ved metodekall:
`set D[1]+3, 2`

Bruk av formell parameter `x` inne i `f`:
`set D[D[0]+3], D[D[D[0]+3]] + 1`

Verdiparametre

Her sender man med en *kopi* av parameteren. (Call by value.)

(Det betyr at hvis den endres i metoden, vil den aktuelle parameteren forbli uendret.)

Overføring av `a`:
`set D[1]+3, D[2]`

Bruk av `x`:
`set D[0]+3, D[D[0]+3] + 1`

Resultatparametre

Dette er det «motsatte» av verdiparametre: Den lokale kopien i metoden kopieres over til den aktuelle parameteren når metoden returnerer. (Call by result.)

Overføring av aktuell parameter `a` *etter* metoderetur:
`set 2, D[D[1]+3]`

Bruk av formell parameter `x` inne i `f`:
`set D[0]+3, D[D[0]+3] + 1`

Verdi-resultatparametre

Dette er en kombinasjon av de to foregående, og effekten er *nesten* den samme som referanseparametre. (Se boken for moteksempler.)

Navneparametre

«Selve uttrykket» sendes over og beregnes på nytt hver gang det brukes. Tekstlig substitusjon. (Call by name.)

```
1  PROCEDURE bytt(x,y)
2  BEGIN
3     INTEGER temp = x;
4     x := y;
5     y := temp;
6  END
```

Se på kallet $bytt(i, a[i])$. Hvis parametrene overføres ved name kan kallet forstås slik:

```
1  PROCEDURE bytt(x,y)
2  BEGIN
3     INTEGER temp = i;
4     i := a[i];
5     a[i] := temp;
6  END
```

som ikke helt er det samme som å bytte om to tall!

Aliasing

To variable (uttrykt i samme programenhet) kalles aliaser hvis deres verdier ligger på samme sted i lageret.

Aliasing kan oppstå ved bruk av

pekere:

```
p1:- new C(...); p2:-p1;
```

referanseoverføring:

```
1  void p(x,y) {
2    x = 1;
3    y = 2;
4  }
5  p(v,v);
```


Aliasing

navneoverføring: (se forrige foil)

ikke-lokale variable:

```
1  int v;
2  void p(x) {
3    x = 4;
4    v++;
5  }
6  p(v);
```

Verdiparametre kan ikke gi opphav til alias.