Programmiersprachen

Vorlesung 2

Zusammenfassung

Georg Ernst Moser

Namen und Scopes

Namen werden in Programmiersprachen durch Deklarationen eingeführt.
Der Scope (also der Bereich in dem ein gewisser Name gültig ist) beginnt am Punkt der Deklaration. Wo dieser endet kann sich jedoch von Sprache zu Sprache stark unterscheiden. Im Allgemeinen existieren:

Static Scoping (lexical scoping)

Lexikalische Programmstruktur → Syntax gibt vor wie die Programmstruktur aussieht.
Entsprechend kann an bestimmten Elementen erkannt werden, wo der Scope endet.
In den meisten static scoping Sprachen kommt eine sogenannte Blockstruktur, bestehend aus einer beginnenden und einer schließenden Klammer, zum Einsatz. Was in diesen Blöcken deklariert wurde gilt bis auch nur bis zum Ende des Blocks (zur schließenden Klammer).

Beispiel:

Drei Variablen werden eingelesen. Im inneren Block wird eine Hilfsvariable z deklariert, welche „zufällig“ genauso heißt wie diese im äußeren Block. Das innere z verdeckt das äußere.

```
#include <stdio.h>

main()
{
int x, y, z;
scanf("%d %d %d", &x, &y, &z);      /* read into x, y, z */

      {                             /* swap x und y */

            int z;                  /* hides z declared in outer block */
            z = x;                  /* z from inner block, x from outer block */
            x = y;                  /* x and y from outer block */
            y = z;                  /* y from outer block, z from inner block
                                          scope of z from inner block ends */
      }

      printf("%d %d %d", x, y, z);
}
```

Dynamic Scoping

Dynamic Scoping wurde vor allem oft in älteren Sprachen verwendet.
Der Scope endet in diesem Fall, wenn etwas neues mit genau demselben Namen deklariert wird. Ein Name gilt dementsprechend bis er durch eine Deklaration “ersetzt” wird (dies kann an einer komplett anderen Stelle im Programm sein).
Ersetzen steht hier im Unterschied zu Verdecken (Verstecken), wie es in Blockstrukturen zum Einsatz kommt: Wird in einem inneren Block ein Name vergeben der gleich einem Namen ist, welcher in einem äußeren Block verwendet wird, verdeckt der innere den äußeren. Es existieren trotzdem beide Namen im selben Scope. Auf den äußeren kann trotzdem manchmal mit gewissen Spracheigenschaften zugegriffen werden.

Beispiel:

```
{     /* Block A */
      int x;
      ...
}
...
```

```
{     /* Block B */
      int x;
      ...
}
...
```

```
{     /* Block C */
      ...
      x = ...;
      ...
}
```

Welches x wird in Block C verwendet?

Je nach Ausführungsreihenfolge wird entweder das x aus Block A oder Block B verwendet.

Static Typing

Statisch typisiert heißt, dass zum Zeitpunkt des Kompilierens jeder Ausdruck bereits einen eindeutig bestimmten Typ hat.

Die Begrifflichkeit kann hier jedoch problematisch sein. In Objektorientierten Sprachen kennt man zwar den Typ, unterscheidet jedoch zwischen deklarierten und dynamischen Typen.

Aus diesem Grund wird zusätzlich zwischen stark und schwach typisiert unterschieden.

Stark typisiert: Der Compiler kann sicherstellen, dass zur Laufzeit keine Typfehler auftreten. Dies ist nicht mit statisch typisiert zu verwechseln, da diese Eigenschaft auch garantiert werden kann ohne dass die Typen bekannt sind (Objektorientierte Sprachen).

Statisch typisiert (Der Einfachheit halber): irgendeine Form der Überprüfung durch den Compiler.

Beispiel in Pascal:

```
var x, y: integer;
c: character;
```

Form der statischen Typisierung. Compiler würde eine Zuweisung von c an x nicht zulassen, da diese unterschiedliche Typen haben.

Explizite Typisierung

Darf nicht mit statischer verwechselt werden. Es gibt nämlich mittlerweile viele Sprachen welche statisch typisiert sind, ohne dass explizit ein Typ angegeben (hingeschrieben) werden muss

Typinferenz(z.B. Haskell):

Der Compiler kann die Typen durch Typinferenz berechnen und sicherstellen dass keine Topkonflikte zur Laufzeit Auftreten.

Fortran:

Es werden zwar keine Typen angegeben, aber diese sind durch Namen bereits zugewiesen: Namen von i bis n sind ganzzahlige Variablen, alle anderen Fließkomma.

Diese Technik wurde angewandt, bevor eine explizite Typannotation erfunden wurde.

Dynamic Typing

Praktisch alle Programmiersprachen sind typisiert. Ausnahmen bilden hier Assembler-Sprachen.

Die meisten Sprachen bei denen man annimmt sie wären nicht typisiert sind dynamisch typisiert.

Die Typen ergeben sich hierbei erst zur Laufzeit. Der Compiler hat infolge dessen keine Verantwortung zur Typüberprüfung, dies geschieht ebenfalls zur Laufzeit.

Sprachen mit dynamic typing werden auch als polymorph bezeichnet. Jeder Name kann unterschiedliche Typen haben (auch stark typisierte objektorientierte Sprachen sind polymorph → ein deklarierter Typ, verschiedene dynamische).

Der Typ einer Variable ändert sich sobald dieser ein neuer Wert zugewiesen wird.

l-Werte

(Name kommt daher, da diese meist auf der linken Seite einer Zuweisung stehen)

Speicheradressen von Variablen

- **Statische Allokation:** Speicher wird bereits vom Compiler allokiert.
- **Dynamische Allokation:** Speicher, genaue Adresse einer Variable wird erst zur Laufzeit bestimmt.
  - **explizit**: zum Beispiel durch das verwenden von „new“ oder „malloc“
  - **implizit**: System sorgt eigenständig für das Anlegen von Speicher

Pointer

Pointer können als l-Werte gesehen werden, welche als r-Werte verwendet werden.

Beispiel:


```
type PI = ^integer;
var pxi: PI;
...
new(pxi);
pxi^ = 0;
```

```
type PPI = ^PI;
var ppxi: PPI;
...
new(ppxi);
ppxi^ = pxi;
```

```
int x = 0;
int* px = &x;
int** ppx = &px;
int y = *px;
**ppx = 5;
```

Links Pascal, rechts C Syntax

Routinen

Sammelbegriff für Methoden, Funktionen, Prozeduren,...

Eine Routine hat wie eine Variable Namen, Scope, Typ, l-Wert und r-Wert.

Der r-Wert ist hierbei die Implementierung, der Rumpf der Routine. Der l-Wert ist die Adresse an welcher diese Implementierung gefunden werden kann.

Normalerweise wird zwischen Kopf und Rumpf der Routine unterschieden.

Kopf: Enthält *Namen* und *Typ*, welche Zusammen die *Signatur* bilden.

Formales Beispiel: fun: T 1 × T 2 × · · · × T n → R [name: Eingangsparameter T1..Tn → Result]. Diese Form der Notation wird in funktionalen und logikorientierten Sprachen verwendet.

Rumpf: Besteht hauptsächlich aus lokalen Variablendeklarationen und ausführbaren Statements.

Routinen sind **first class Objects**, wenn Parameter auch Routinen enthalten können. Routinen können in diesem Fall so verwendet werden wie alle anderen Daten auch.

Der Umgang mit diesen first class objects ist jedoch schwierig, weswegen auch gern darauf verzichtet wird, Funktionen zu first class objects zu machen (In funktionalen Sprachen sind Funktionen jedoch so wichtig).

Aufruf: Spezifiziert formelle Parameter mit den aktuellen Parametern

Function and Pointer to Function in C

```
int sum(int n)
{
      int i, s;
      s = 0;
      for (i = 1; i <= n; i++)
            s = s + 1;
      return s;
}
```

```
int i;
int (*ps)(int);
...
ps = &sum;
i = (*ps)(5);   /* i = sum(5); */
```

Zeiger auf Funktionen können an Variablen zugewiesen werden...

In gewisser Weise werden Funktionen also wie first class objects verwendet?

Dies stimmt so nicht ganz.Funktionen können nämlich nicht als Rückgabeparamter verwendet werden (Dies Problem liegt hier beim Abbau des Stacks, genauere Erklärung folgt später).

Wie kann dieser Ausdruck gelesen werden: int (*ps)(int) ?

Man beginnt in der Mitte bei dem Namen welcher deklariert wird und geht dann zuerst nach rechts und wenn es rechts nicht mehr weiter geht nach links und dann eine Klammerebene nach außen.

int (*ps)(int)

ps ist ein Zeiger auf eine Funktion, welche eine ganze Zahl als Parameter nimmt und eine ganze Zahl zurückliefert.

ps = ∑ // hier muss dereferenziert werden

i = (*ps)(5) //hier muss um 5 ein Klammer gesetzt werden

Deklaration vs. Definition

Deklaration: Ein Name wird für etwas eingeführt

Definition: Es wird auch angegeben wofür ein Name stehen soll

```
int A(int x, int y);              /* declaration of A */
float B(int z)                    /* definition of B */
{
      int w, u;
      ...
      w = A(z, u);                /* A usable here */
      ...
};

int A(int x, int y)               /* definition of A */
{
      float t;
      ...
      t = B(x);                   /* B usable here */
      ...
};
```

Wir sind es gewohnt Deklarationen von oben nach unten zu lesen. In diesem Beispiel muss alles zuerst deklariert werden, bevor es verwendet werden kann.

Wenn in A die Funktion B aufgerufen werden soll und in B A, kommt es zu einem Problem: Zum Zeitpunkt des Aufrufs von A ist A noch nicht definiert (und dementsprechend auch nicht deklariert). Aus diesem Grund muss eine Deklaration von A eingeführt werden, die getrennt von der Definition ist. Auf diese Weise kann der Zyklus aufgelöst werden

Ähnlich: Auch durch die Trennung von Interfaces und Klassen können Zyklen aufgelöst werden.

Invocation of Routines

Wir sprechen für gewöhnlich von der Instanz einer Routine wenn wir einen gewissen Aufruf meinen.

Eine **Instanz** besteht aus einem

- Codesegment (dem Segment, dem der Rumpf entspricht)
- Activation Record (Wo Stacks vorhanden auch auch Stackframe genannt): In diesem sind vor allem lokale Variablen als auch andere Verwaltungsdaten gespeichert.

Lokale Umgebung: hauptsächlich der Activation Record

Nichtlokale Umgebung: alle anderen sichtbaren Objekte (Variablen,...)

Warum spricht man beim Gegenstück zu lokalen Variablen nicht von globalen Variablen? Globales Environment gehört zum nichtlokalen, aber für alle sichtbar. Die nichtlokale Umgebung ist wesentlich größer. Es gehören auch lokale Umgebungen von andern Umgebungen auf welche gerade Zugriff besteht dazu.

Rekursion:

Schwierig in der Implementierung einer Programmiersprache. Deswegen waren ältere Sprachen wie Fortran nicht rekursiv. Fortran wird zum Beispiel aus diesem Grund heute noch verwendet, da es sehr performant ist (number crunching).

Bei Rekursion muss für jeden Funktionsaufruf ein eigener Activation Record erstellt werden. Es existieren folglich mehrere Activation Records für dieselbe Routine zur gleichen Zeit. Dies macht eine Form des dynamischen Bindens zwischen den Codesegmenten erforderlich.

Formale Parameter: Parameter welche im Kopf einer Routine deklariert sind.

Eigentliche Parameter (actual parameters): Parameter welche beim Aufruf übergeben werden.

Parameter Binding

(Beispiele in Ada)

```
procedure Example(A: T1; B: T2 := W; C: T3);
```

T2 hat default Belegung: Gibt man im Aufruf kein Argument an, wird W genommen.

Bindung über

- **Position**

  ```
  Example(X, Y, Z);
  ```

  X wird an A, Y an B und Z an C gebunden (begingt durch deren Position)

- **Namen**

  ```
  Example(C => Z, A => X, B => Y);
  ```

  Die Reihenfolge kann hier variiert werden (muss nicht gemerkt werden). Für B könnte zum Beispiel gar kein Parameter angegeben werden. Die default Belegung würde in diesem Fall verwendet werden (diese kann sonnst nur für den letzten Parameter zur Anwendung kommen).

- **Gemischte Verwendung**

  ```
  Example(X, C => Z)
  ```

  A und B werden über Position gebunden, C über den Namen.

Generic Routine

Beispiel: Template Funktion, welche 2 Werte vertauscht.

```
template <class T> void swap(T& a, T& b)
/* generic routine in C++ swaps a und b */
{
      T temp = a;
      a = b;
      b = temp;
}
int i, j;                                   float f, g;
...                                         ...
swap(i, j);                                 swap(f, g);
```

Der Typparameter wird hierbei wie ein regulärer Typ verwendet. Trotzdem sind diese nicht dasselbe.

Bei generischen Methoden wird immer Typinferenz verwendet (Ausnahme Ada).

Overloading und Aliasing

Overloading: Ein Name, welcher für unterschiedliches stehen kann.

`int i, j, k;`
`float a, b, c;`
`...`
`i = j + k;`
`a = b + c;`
`a = b + c + b();`
`a = b() + c + b(i);`
	Beispiel:
	„Plus“ Operator wird für ganze Zahlen und Fließkommazahlen überladen. Der deklarierte Typ entscheidet welche Funktion zur Anwendung kommt (weiterer Grund warum Typdeklarationen eingeführt wurden).
	B ist sowohl Variable als auch Funktion. Beide können aber durch die Art der Verwendung unterschieden werden.
	Überladen ist ein Syntaktische Vereinfachung, damit nicht so viele Symbole benötigt werden.

Aliasing: Mehrere Namen, welche auf dasselbe verweisen.

Imperative Sprachen würden ohne Aliasing schwer auskommen.

`int x = 0;`
`int *i = &x;`
`int *j = &x;`
`...`
`*i = 10;`
	Beispiel:
	Sowohl *i als auch *j verweise n auf x (zuerst 0, dann durch Zuweisung 10). Beide Namen führen zur selben Speicherzelle.

Implementation of Programming Languages

SIMPLESEM

Verwendung eines sehr einfachen pseudo Assemblers.

Es gibt 2 Arten von **Speicher**:

- Datenspeicher (D)
- Code Speicher (C)

Instruction Pointer: Dieser zeigt auf die aktuelle Stelle im Programm

Instruktionen:

Instruktion	**Bedeutung**
`set target, source`	*target* kann immer nur Datenspeicher sein! *source* kann Daten oder Code Speicher sein.
`set 10, D[20]`	*D[Zelle]* steht für Datenspeicher
`set 15, read`	in Zelle 15 speichere das was aus der Standardeingabe gelesen wurde
`set write, D[50]`	*write* ist die Standardausgabe
`set 99, D[15]+D[33*C[41]]-8`	beliebig komplexe Ausdrücke möglich
`set D[10], D[20]`	In *D[10]* schreibe den Inhalt aus *D[20]*
`jump 47`	Sprung zu Zeile 47
`jumpt 47, D[3]>D[8]`	Bedingter Jump
`jump D[13]`	inderekter Jump: Springe an Zelle die in *D[13]* steht
`halt`	Programmende

Beispiel (in C ähnlicher Sprache):

```
main()
{
      int i, j;            code memory                data memory
                           +------------------------  +-------------
      get(i, j);           |0 set 0, read             |0 cell for i
                           |1 set 1, read             |1 cell for j
      while (i != j)       |2 jumpt 8, D[0] = D[1]
            if (i > j)     |3 jumpt 6, D[0] <= D[1]
                  i -= j;  |4 set 0, D[0] - D[1]
            else           |5 jump 7
                  j -= i;  |6 set 1, D[1] - D[0]
                           |7 jump 2
      print(i);            |8 set write, D[0]
}                          |9 halt
```

Übersetzung:

Compiler reserviert für i und j Speicherzellen.

Einlesen der Variablen.

While Schleife: Bedingung wird umgedreht (sind wir schon am Ende der Schleife?)

if i == j dann springe ans Ende der Schleife

if Abfrage: Zweig wird übersprungen falls Bedienung nicht stimmt (direkt in else Zweig).

Am Ende der Schleife springe an Anfang der Schleife.

Optimierungen möglich, aber wegen Einfachheit weggelassen.

Routinen

```
int i=1, j=2, k=3;
```

```
alpha()
{
int i=4, l=5;
...
i += k + l;
...
};
```

```
beta()
{
int k=6;
...
i = j + k;
alpha();
...
};
```

```
main()
{
...
beta();
...
}
```

Übersetzung:

```
code memory
+-----------------------------------
|     ...               // main
|014  set 6, 16
|015  jump 100
|     ...
|049  halt
|     ...               // alpha
|058 set 4, D[4]+D[2]+D[5]
|     ...
|099 jump D[3]
|     ...               // beta
|122 set 0, D[1]+D[7]
|123 set 3, 125
|124 jump 50
|     ...
|149 jump D[6]
|     ...               // ...
```

```
data memory
+--------------------
|0    8     //i         global
|1    2     //j
|2    3     //k
|3    125   //ret. alpha
|4    12    //i
|5    5     //l
|6    16    //ret. beta
|7    6     //k
| ...



     ip = 59
```

Im Datenspeicher wird an bestimmten Stellen Platz für Variablen reserviert. Oben globale dann Speicher für die Routinen alpha und beta. Es wird auch Platz für einen return pointer gelassen, welcher benötigt wird um nach einem Methodenaufruf zurück zur Aufrufstelle zu springen.

Main:

an Stelle 6 (return pointer für beta) setze die Nummer 16 (Stelle an die Nach dem Aufruf von beta gesprungen werden soll).

Springe an Stelle 100 (Beginn von beta)

Beta:

rufe alpha auf. Nach Aufruf springe an Adresse, die der return pointer enthält.

Annahme: Activation Records können fix an einzelne Routinen zugeordnet werden. Es gibt folglich keine Rekursion in dieser Sprache: Beim Aufruf einer Routine in derselben Routine würde der return pointer, welcher eigentlich noch gebraucht wird überschrieben werden.

Seperate Übersetzung

```
/* file 1 */
int i=1, j=2, k=3;
extern beta();
main()
{
      ...
      beta();
      ...
}
```

```
/* file 2 */
extern int k;
alpha()
{
      int i=4, l=5;
      ...
      i += k + l;
      ...
}
```

```
/* file 3 */
extern int i,j;
extern alpha();
beta()
{
      int k = 6;
      ...
      i = j + k;
      alpha();
      ...
}
```

Compiler übersetzt zuerst File1, dann File2, dann File3. Dies ist ohne Komplikationen möglich, da die Blöcke unabhängig voneinander sind.

Am Ende müssen lediglich die Adressen übergeben werden (Aufgabe des Linkers)

Rekursion und Rückgabe von Werten

```
int n;
int fact()
{
      int loc;
      if (n > 1) {
            loc = n--;
            return loc * fact();
      }
      else
            return 1;
}
```

```
main()
{
      get(n);
      if (n >= 0)
            print(fact());
      else
            print("input error");
}
```

Speicherorganisation:

Wie können mehre Activation Records für eine Routine realisiert werden: Einführung eines Stacks.

Gesamter Datenspeicher wird als ein Stack organisiert:

Ganz oben finden sich in fixen Speicherzellen die variablen CURRENT und FREE.

```
data memory
+-----------------------------------------------------
|0 B        //CURRENT
|1 C        //FREE
|     ...               stack grows downwards
|                       holds activation records
|
|A    ...   //return pointer (caller’s activation record)
|     ...   //dynamic link
|     ...   //local variables
|
|B    ...   //return pointer (current activation record)
|     A     //dynamic link
|     ...   //local variables
|
|C    ...   //free memory
```

CURRENT: Zeiger auf den aktuellen Activation Record (Daten der gerade aktiven Routine)

FREE: Zeiger an das Ende des Stacks (Ab diesem beginnt der freie Speicher)

Aufbau des Activation Records:

- Return Pointer
- lokale Variablen
- dynamic link: Bei Rückkehr aus einer Methode muss auch an die richtige Stelle im Stack gesprungen werden. Enthält den Activation Record zu dem gesprungen werden soll (Anfang des Activation Records der Methode welche die aktuell aufgerufen hatte). Die *dynamic chain*, also die Kette der dynamic links bildet dynamisch Aufrufreihenfolge ab.

  Der dynamic link wird notwendig, falls für eine Routine mehr als ein Activation Record existieren kann (Rekursion). Andernfalls ist ein Return Pointer ausreichend.
- formale Parameter (Parameter die an Funktion übergeben werden)

Aufruf und Rückgabe von Werten:

`set 1, D[1]+1`
`set D[1], ip+4`
`set D[1]+1, D[0]`
`set 0, D[1]`
`set 1, D[1]+AR`
`jump addr`
	`allocate memory for result`
	`set return pointer`
	`set dynamic link`
	`set CURRENT`
	`set FREE`
	`jump to routine`

`set 1, D[0]`
`set 0, D[D[0]+1]`
`jump D[D[1]]`
	`set FREE`
	`set CURRENT`
	`jump to stored return pointer`

Das Ergebnis (Rückgabewert) einer Routine wird im Activation Record ebendieser abgelegt. Am Ende des AR ist eine Zelle dafür reserviert.

Aufruf:

FREE Zeiger wird um 1 erhöht (Activation Record um 1 vergrößert)

Stelle des Neuen Activation Record ist der momentane FREE pointer.

Dynamic link: neuer DL soll genau die Stelle des aktuellen AR sein.

Neuer Current pointer ist der alte FREE Pointer.

Neuer FREE Pointer ist der alte FREE pointer + größe des AR.

Rücksprung:

FREE pointer = Anfang des aktuellen AR
CURRENT pointer = DL des AR der noch aktiv ist.

Ergebnis kann mit CURRENT pointer – 1 ausgelesen werden. Auf diesen Teil können sowohl Aufrufer als auch aufgerufene Methode zugreifen.

Nested Loops

```
int f() {                                  | activation record of f
      int x,y,w;                    //1    | +-----------------------------
      while(...) {                         | | return pointer
            int x,z;                //2    | | dynamic link
            while(...) {                   | | x in //1
                  int y; ...        //3    | | y in //1
            }                              | | w in //1
            if(...) {                      | | x in //2, a in //5
                  int x,w; ...      //4    | | z in //2, b in //5
            }                              | | y in //3, x in //4, c in //5
      }                                    | | w in //4, d in //5
      if(...) {                            | +-----------------------------
            int a,b,c,d;            //5
                  ...
      }
}
```

Einfache Lösung:

Wenn in der Schachtelung **nur Variablendeklarationen** vorkommen (Java,C,..) kann dieselbe Speicheradresse für mehre Variablen reserviert werden, da diese niemals gleichzeitig gebraucht werden.

Block 1: was hier deklariert wurde ist bis zum Ende sichtbar → belegen jeweils eine Speicheradresse.

Block 2: Die Variablen welche in der while Schleife deklariert wurden werden ab dem if statement (Block 5) nicht mehr benötigt. Daher können dieselben Speicherzellen, welche vorhin die Variablen des while Blocks enthalten haben jetzt für die des If Blocks verwendet werden.

Block 3: gleich wie in Block 2

Nested Routines and Static Link

```
int x,y,z;                          sketch of run-time stack
      void f1() {                   +---------------------------------
            int t,u;                |A global environment (x,y,z),
            void f2() {             |      no dyn.link, stat.link needed
                  int x,w;          |B act.record for f1 (t,u) with
                  void f3() {       |     dyn.link: A, stat.link: A
                        int y,w,t;  |C act.record for f2 (x,w) with
                        ...         |     dyn.link: B, stat.link: B
                  }                 |D act.record for f3 (y,w,t) with
                  ...               |     dyn.link: C, stat.link: C
            }                       |E act.record for f2 (x,w) with
            ...                     |     dyn.link: D, stat.link: B !!
      }                             |
```

Wie können die Variablen auf welche zugegriffen werden kann gefunden werden?

In f3() zum Beispiel sind die Variablen von f3() sichtbar, es kann aber auch auf die Variablen von f2() zugegriffen werden. ABER: Welche Instanz, welcher Activation Record von f2()? Es muss nicht sein, dass f2() f3() aufruft! Es kann auch f3() f2() aufrufen,... Wie können hier die AR gefunden werden?

Aufbau des Stacks:

Globales Environment [A]: gehört nicht zu einer einzelnen Funktion

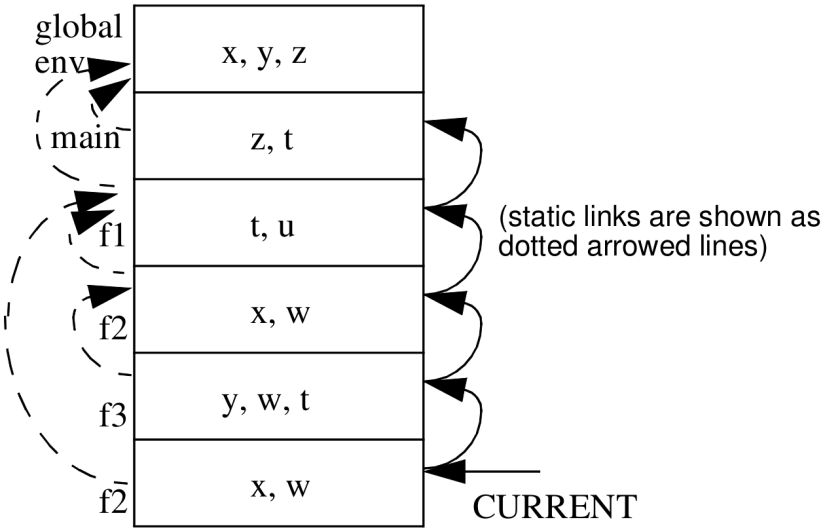
Beim Aufruf von f1() [B] wird Platz für t, u und den dynamic link (Ziel dessen ist in erster Ebene nicht unbedingt von Bedeutung, meist wird aber das globale Environment als Ziel angegeben) benötigt.

Zusätzlich wird ein **static link** benötigt: Dieser macht klar wie die statische Umgebung einer Funktion aussieht. Er zeigt auf den AR der übergeordneten Funktion welche die aktuelle Funktion statisch im Programmcode umgibt.

Es folgt der Aufruf von f2() [C]: dynamic link: B, static link: B

Es folgt der Aufruf von f3() [D]: dynamic link: C, static link: C

Dann wird innerhalb von f3() f2() aufgerufen [E]: dynamic link: D (Aufrufer) , static link: B (Activation Record des statischen Umgebung von f3())



Grafische Veranschaulichung des obigen Programmbeispiels

Nested Routines in Programmiersprachen

Java, C++, C#, VB und Eiffel unterstützen geschachtelten Funktionen nicht direkt (es wird dementsprechend kein static link benötigt).

In Java zum Beispiel können geschachtelten Funktionen jedoch auf folgende Arten simuliert werden:

- Verwendung von Lambda Ausdrücken (Lambda Ausdrücke erlauben den Zugriff auf Variablen im Scope der umgebenden Funktion).
- Verwendung einer anonymen Klassen, welche genau eine Funktion enthaltet

Computing Frame Pointer

Ein Frame-Pointer eines Methodenaufrufs ist eine Kopie des Stack-Pointers, vor dem Aufruf der Methode.

Jede Variable wird durch ein Paar <d,o> ausgedrückt, wobei

d: Entspricht der Distance zwischen der Umgebung in der wir uns gerade befinden und der auf die wir zugreifen wollen (wie viele Ebenen weiter draußen).

Beispiel (siehe Code auf vorheriger Seite): Wenn man in f3() auf eine Variable von f1() zugreifen will, liegen diese 2 Ebenen weiter draußen.

o: offset der Variablen innerhalb des Activation Records.

Der Framepointer fp in Abhängigkeit von der Distanz d, kann mit folgender Formel berechnet werden:

```
fp(d) = if d=0 then D[0] else D[fp(d−1)+stat.link.offset]
```

Wenn die Distanz 0 ist → lokale Umgebungen

Sonnst: Berechne fp(d—1) rekursiv und addiere dazu den offset des static links im Activation Record.

Beispiel:

```
fp(0) = D[0],
fp(1) = D[D[0]+stat.link.offset],
fp(2) = D[D[D[0]+stat.link.offset]+stat.link.offset]
```

Die eigentliche Adresse an welche die Variable gebunden ist entspricht dann:

```
D[fp(d)+o]
```

Dieses Verfahren erfordert viele Speicherzugriffe und ist dementsprechend relativ teuer. Aus diesem Grund unterstützen moderne Programmiersprachen oft keine verschachtelten Routinen mehr.

Routinenaufruf mit static links

Wurde übersprungen

Dynamische Arrays

Dynamische Arrays sind Arrays, deren Größe zur Zeit des Kompilierens nicht bekannt ist. Wurden ursprünglich in Ada eingeführt und sind mittlerweile fast überall verfügbar.

Bei deren Erstellung wird vom Compiler im Activation Record nur Speicher für einen Descriptor des Arrays festgelegt. Das eigentliche Array wird trotzdem im AR allokiert.

Zum Zeitpunkt der Deklaration der Variablen ist die Größe bereits bekannt. Der Activation Record wird dementsprechend vergrößert. Dies geschieht dynamisch und kann gegebenenfalls auch öfters hintereinander durchgeführt werden.

Der Descriptor enthält die Größe oder Anfangs- und Endindex und die Adresse des eigentlichen Arrays.

Voraussetzung für diese Verfahren ist natürlich, dass der Activation Record eine variable Größe hat.

Dynamic Scoping

```
void sub2() {
      declare x;
      ...
}
void sub1() {
      declare y;
      sub2();
      ...
}
void main() {
      declare x,y,z;
      sub1();
      sub2();
}
```

```
sketch of run-time stack
+-------------------------------------
|A link     = none      (act.record for main)
|     x     = ...
|     y     = ...
|     z     = ...
|B link     = A         (act.record for sub1)
|     y     = ...
|C link     = B         (act.record for sub2)
|     z     = ...


dynamic search for name along link
```

In der main() Funktion werden einige Variablen deklariert und dann die beiden Funktionen sub1() und sub2() aufgerufen. In beiden Unterfunktionen werden Variablen deklariert. In sub1() wird dementsprechend das y aus main() ersetzt und in sub2() das x.

Im Stack werden um immer die richtigen Variablen auffinden zu können, Name und Inhalt (meist nur Referenz darauf → Wer am Heap) abgespeichert. Da immer Paare abgelegt werden wird mehr Speicher benötigt. Zusätzlich wird weiterhin ein Link (analog zu dynamic link) abgespeichert. Beim Suchen nach Variablen wird zuerst im eigenen Activation Record begonnen und dann dem „Link“ nach außen gefolgt.

Heap

Alle Objekte auf welche von mehreren Orten zugegriffen werden kann, werden auf den Heap gelegt.

Der Stack wird jedes mal abgebaut, nachdem aus einer Routine zurückgekehrt wird. Der Heap bleibt bestehen.

Klassische Speicheraufteilung:

```
data memory
+-------------------------------------------------
| stack:    grows downwards,
|            holds activation records removed on return
|
|
| heap:     grows upwards,
|           holds data alive until program termination
+-------------------------------------------------
```

Wenn Stack und Heap zusammenstoßen ist der Speicher aus.

Aktuell etwas komplizierter:

Es existieren meist mehrere Threads in einem Programm. Jeder dieser Threads hat einen eigenen Stack mit einer bestimmten Größe am Heap reserviert.

Es gibt auch Sprachen die über mehr als einen Heap verfügen.

Parameter Passing

- **call by reference**: Eine Adresse der Speicherzelle wird übergeben. Die Aufgerufene Methode kann auch in die übergebene Adresse schreiben.
- **call by copy**
  - **call by value**: Kopiert den Wert in den Activation Record der aufgerufenen Methode
  - **call by result**: Die aufgerufene Methode kopiert den Wert in den AR des Aufrufers um diesen zurückzugeben.
  - **call by value-result**: Kombination aus call by value und call by result. Variable wird zuerst vom Aufrufer in die aufgerufene Methode kopiert und am Ende der Methode von der aufgerufen Methode wieder zum Aufrufer zurückkopiert.
- **call by name**: Jedes Vorkommen des formalen Parameters wird durch den aktuellen Parameter ersetzt.

Implementation of Call by Reference

Bei Call by Reference muss vom Aufrufer unterschieden werden welche Art von Daten übergeben wird:

- Beim Übergeben einer normalen Variable muss die Adresse dieser übergeben werden.
- Wenn der Wert im Aufrufer jedoch schon ein call by reference Wert ist (ein Pointer) muss der Inhalt dessen übergeben werden.

Call by Reference vs. Value Result

Performance:

Bei der Verwendung von Call by Value muss einmal weniger dereferenziert werden. Wenn allerdings mehrmals (oder sehr viel) kopiert werden muss ist der Performance Vorteil von Call by Value hinfällig.

Beispiel:

2 formale Parameter als Aliases

Aktuelle Parameter: a[i] und a[j] (Absichtlich Array Elemente gewählt, da Vergleich auf Gleichheit nicht ohne weiteres möglich)

Formale Parameter: x und y

Routine:

```
x = 0;
y++;
```

- **Call by reference**:

  Anhand der Adresse von a[i] wir die Speicherzelle auf 0 gesetzt. In der nächsten Instruktion wird zufällig, da a[j] auch auf diese zeigt, dieselbe Speicherzelle um 1 erhöht (x=y=1).

- **Call by Value Result**:

  Der Wert von a[i] (xv) wird in x kopiert (x = xv) und er Wert von a[j] (yv) wird in y kopiert (y = yv). An x wird 0 zugewiesen (x=0). y wird um 1 erhöht (y = yv +1). Im Unterschied zu call by reference ist hier am Ende der Routine x != y.

  Anschließend werden die Ergebnisse zurückgeschrieben.

Call by Name

Unter call by name versteht man was in der Mathematik „Ersetzung“ heißt. Der formale Parameter wird durch den aktuellen ersetzt. Kommt es zu einem „Clash“ (ein Namenskonflikt unter den Parametern) wird umbenannt.

Abseits der Verwendung in funktionalen Sprachen, ist dieses Verfahren jedoch nicht so gut zur Übergabe von Parametern geeignet wie ursprünglich (call by name war default in Algol 60, call by value die Alternative) gedacht:

Beispiel 1: Vertauschen zweier Integer Werte

`swap(int a, b) {`
`  int temp = a;`
`  a = b;`
`  b = temp;`
`};`
`  ...`
`  swap(i, a[i]);`
`  …`
	Es sollen a[i] und i verstaucht werden. Es wird mit call by name gearbeitet.
	Beim Aufruf von swap(i , a[i]) werden alle formalen Parameter durch die aktuellen ersetzt:
	`int temp = i;`
	`i = a[i];`
	`a[i] = temp;  //ACHTUNG`
	Es wird bei dieser Zuweisung der falsche Index verwendet, da i mittlerweile geändert wurde.
	In diesem Fall verhält sich die Routine nicht so wie vom Programmierer erwartet. Das Erstellen einer swap-Funktion war in Algol 60 dementsprechend schwieriger als gedacht.

Beispiel 2: Eine abgeschwächte Variante von call by name wie sie noch immer verwendet wird.

`int c; /* global variable */`
 	
`swap(int a, b) {`
`  int temp = a; a = b;`
`  b = temp; c++;`
`};`
 	
`y() {`
`  int c, d;`
`  swap(c, d);`
`};`
	Parameterübergabe in Makros.
	Es sollen wieder 2 Variablen vertauscht werden. Allerdings wird diesmal zusätzlich eine globale Variable c um 1 erhöht.
	Echtes call by name kann in diesem Fall die beiden c's (das globale c in swap und das übergebene c von y) unterscheiden. Durch Umbenennung wird sichergestellt, dass es zu keinen Namenskonflikten kommt.
	Die abgespeckte Variante die in Makros zum Einsatz kommt, kann zwischen den beiden jedoch nicht unterscheiden.
	`temp = c; c = d;`
	`d = temp;`
	`c++;  !!`
	Auch kann es bei dieser Variante erforderlich sein zuerst Klammern um den Wert, welcher übergebenden werden soll, zu setzten („Echtes“ call by name übernimmt dies für den

	Programmierer).

Routine as Parameter

Eine Routine wird an eine andere übergeben.

Ansatz: Wie in C einen Pointer auf die Routine übergeben.

Problem: Geschachtelte Funktionen. Für diese wird ein static link benötigt.

Es muss zusätzlich zum Zeiger auf die Routine auch der static link übergeben werden (beschreibt die Umgebung der aufrufenden Routine)

```
1  int u, v;
2  void a()
3  {
4        int y;
5        ...
6  };
```

```
7  void b(routine x)
8  {
9        int u, v, y;
10       void c() {
11             ...
12             y = ...;
13             ...
14       };
15       x();
16       b(c);
17       ...
18 }
```

```
19  void main()
20  {
21        b(a);
22  };
```

Alternative Lambda Ausdrücke. Diese werden direkt in der Umgebung ausgeführt (eingebaut).