

5 Speicherbelegung für Variablen

Ziel:

Ordne jeder Variablen x **statisch**, d. h. zur Übersetzungszeit, eine feste (Relativ-)Adresse ρx zu!

Annahmen:

- Variablen von Basistypen wie **int**, ... erhalten eine Speicherzelle.
- Variablen werden in der Reihenfolge im Speicher abgelegt, wie sie deklariert werden, und zwar ab Adresse 1.

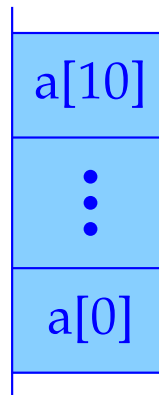
Folglich erhalten wir für die Deklaration $d \equiv t_1 x_1; \dots t_k x_k$ (t_i einfach) die Adress-Umgebung ρ mit

$$\rho x_i = i, \quad i = 1, \dots, k$$

5.1 Felder

Beispiel: `int [11] a;`

Das Feld `a` enthält 11 Elemente und benötigt darum 11 Zellen.
 ρa ist die Adresse des Elements `a[0]`.



Notwendig ist eine Funktion `sizeof` (hier: $|\cdot|$), die den Platzbedarf eines Typs berechnet:

$$|t| = \begin{cases} 1 & \text{falls } t \text{ einfach} \\ k \cdot |t'| & \text{falls } t \equiv t'[k] \end{cases}$$

Dann ergibt sich für die Deklaration $d \equiv t_1 x_1; \dots t_k x_k;$

$$\begin{aligned} \rho x_1 &= 1 \\ \rho x_i &= \rho x_{i-1} + |t_{i-1}| && \text{für } i > 1 \end{aligned}$$

Weil $|\cdot|$ zur Übersetzungszeit berechnet werden kann, kann dann auch ρ zur Übersetzungszeit berechnet werden.

Aufgabe:

Erweitere `codeL` und `codeR` auf Ausdrücke mit indizierten Feldzugriffen.

Sei $t[c] a;$ die Deklaration eines Feldes a .

Um die Anfangsadresse der Datenstruktur $a[i]$ zu bestimmen, müssen wir $\rho a + |t| * (R\text{-Wert von } i)$ ausrechnen. Folglich:

```
codeL a[e] ρ = loadc (ρ a)
               codeR e ρ
               loadc |t|
               mul
               add
```

... oder allgemeiner:

$$\text{code}_L e_1[e_2] \rho = \text{code}_R e_1 \rho$$

$$\text{code}_R e_2 \rho$$

$$\text{loadc } |t|$$

$$\text{mul}$$

$$\text{add}$$

Bemerkung:

- In **C** ist ein Feld ein **Zeiger**. Ein deklariertes Feld a ist eine **Zeiger-Konstante**, deren R-Wert die Anfangsadresse des Feldes ist.
- Formal setzen wir für ein Feld e : $\text{code}_R e \rho = \text{code}_L e \rho$
- In **C** sind äquivalent (als L-Werte):

$$2[a] \quad a[2] \quad a + 2$$

5.2 Strukturen

In **Modula** heißen Strukturen **Records**.

Vereinfachung:

Komponenten-Namen werden nicht anderweitig verwendet.
Alternativ könnte man zu jedem Struktur-Typ st eine separate
Komponenten-Umgebung ρ_{st} verwalten :-)

Sei `struct { int a; int b; } x;` Teil einer Deklarationsliste.

- x erhält die erste freie Zelle des Platzes für die Struktur als Relativ-Adresse.
- Für die Komponenten vergeben wir Adressen **relativ** zum Anfang der Struktur, hier $a \mapsto 0, b \mapsto 1$.

Sei allgemein $t \equiv \mathbf{struct} \{t_1 c_1; \dots t_k c_k\}$. Dann ist

$$\begin{aligned} |t| &= \sum_{i=1}^k |t_i| \\ \rho c_1 &= 0 \quad \text{und} \\ \rho c_i &= \rho c_{i-1} + |t_{i-1}| \quad \text{für } i > 1 \end{aligned}$$

Damit erhalten wir:

$$\begin{aligned} \mathbf{code}_L(e.c) \rho &= \mathbf{code}_L e \rho \\ &\quad \mathbf{loadc}(\rho c) \\ &\quad \mathbf{add} \end{aligned}$$

Beispiel:

Sei `struct { int a; int b; } x;` mit $\rho = \{x \mapsto 13, a \mapsto 0, b \mapsto 1\}$.

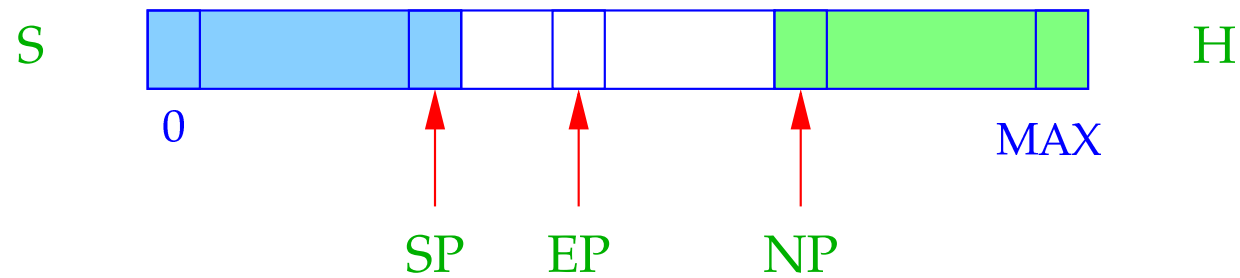
Dann ist

$$\text{code}_L(x.b) \rho = \begin{array}{l} \text{loadc } 13 \\ \text{loadc } 1 \\ \text{add} \end{array}$$

6 Zeiger und dynamische Speicherverwaltung

Zeiger (Pointer) gestatten den Zugriff auf anonyme, dynamisch erzeugte Datenelemente, deren Lebenszeit nicht dem **LIFO**-Prinzip unterworfen ist.

\implies Wir benötigen eine weitere potentiell beliebig große Datenstruktur **H** – den **Heap** (bzw. die **Halde**):



NP $\hat{=}$ **New Pointer**; zeigt auf unterste belegte Haldenzelle.

EP $\hat{=}$ **Extreme Pointer**; zeigt auf die Zelle, auf die der **SP** maximal zeigen kann (innerhalb der aktuellen Funktion).

Idee dabei:

- Chaos entsteht, wenn Stack und Heap sich überschneiden (**Stack Overflow**).
- Eine Überschneidung kann bei jeder Erhöhung von **SP**, bzw. jeder Erniedrigung des **NP** eintreten.
- **EP** erspart uns die Überprüfungen auf Überschneidung bei den Stackoperationen :-)
- Die Überprüfungen bei Heap-Allokationen bleiben erhalten :-).

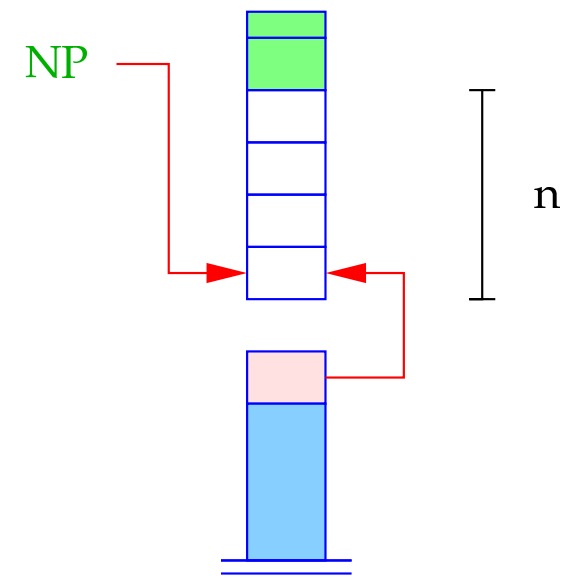
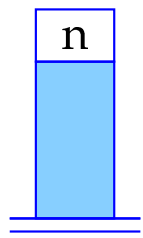
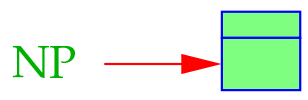
Mit Zeiger (-Werten) rechnen, heißt in der Lage zu sein,

- Zeiger zu **erzeugen**, d.h. Zeiger auf Speicherzellen zu setzen; sowie
- Zeiger zu **dereferenzieren**, d. h. durch Zeiger auf die Werte von Speicherzellen zugreifen.

Es gibt zwei Arten, Zeiger zu erzeugen:

- (1) Ein Aufruf von **malloc** liefert einen Zeiger auf eine Heap-Zelle:

$$\text{code}_R \text{ malloc } (e) \rho = \text{code}_R e \rho \text{ new}$$



```
if (NP - S[SP] ≤ EP)
    S[SP] = NULL;
else {
    NP = NP - S[SP];
    S[SP] = NP;
}
```

- NULL ist eine spezielle Zeigerkonstante (etwa 0 :-)
- Im Falle einer Kollision von Stack und Heap wird der NULL-Zeiger zurückgeliefert.

- (2) Die Anwendung des Adressoperators `&` liefert einen **Zeiger** auf eine Variable, d. h. deren Adresse ($\hat{=}$ **L-Wert**). Deshalb:

$$\text{code}_R (\&e) \rho = \text{code}_L e \rho$$

Dereferenzieren von Zeigern:

Die Anwendung des Operators `*` auf den Ausdruck e liefert den **Inhalt** der Speicherzelle, deren Adresse der R-Wert von e ist:

$$\text{code}_L (*e) \rho = \text{code}_R e \rho$$

Beispiel:

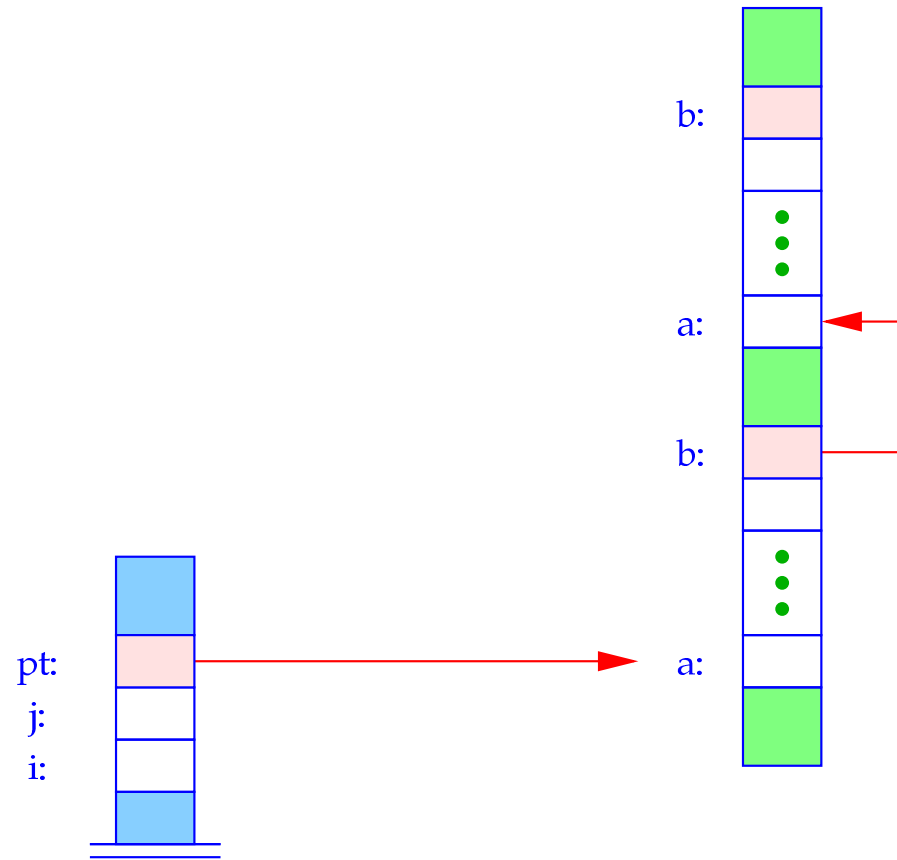
Betrachte für

```
struct t { int a[7]; struct t *b; };  
int i, j;  
struct t *pt;
```

den Ausdruck $e \equiv ((pt \rightarrow b) \rightarrow a)[i + 1]$

Wegen $e \rightarrow a \equiv (*e).a$ gilt:

$$\text{code}_L(e \rightarrow a) \rho = \text{code}_R e \rho$$
$$\text{loadc}(\rho a)$$
$$\text{add}$$



Sei $\rho = \{i \mapsto 1, j \mapsto 2, pt \mapsto 3, a \mapsto 0, b \mapsto 7\}$. Dann ist:

$$\begin{array}{lcl}
 \text{code}_L e \rho & = & \text{code}_R ((pt \rightarrow b) \rightarrow a) \rho \\
 & & \text{code}_R (i + 1) \rho \\
 & & \text{loadc } 1 \\
 & & \text{mul} \\
 & & \text{add} \\
 & = & \text{code}_R ((pt \rightarrow b) \rightarrow a) \rho \\
 & & \text{loada } 1 \\
 & & \text{loadc } 1 \\
 & & \text{add} \\
 & & \text{loadc } 1 \\
 & & \text{mul} \\
 & & \text{add}
 \end{array}$$

Für Felder ist der R-Wert gleich dem L-Wert. Deshalb erhalten wir:

$$\begin{aligned}
 \text{code}_R((pt \rightarrow b) \rightarrow a) \rho &= \text{code}_R(pt \rightarrow b) \rho &= & \begin{array}{l} \text{loada 3} \\ \text{loadc 7} \\ \text{add} \\ \text{load} \\ \text{loadc 0} \\ \text{add} \end{array}
 \end{aligned}$$

Damit ergibt sich insgesamt die Folge:

loada 3	load	loada 1	loadc 1
loadc 7	loadc 0	loadc 1	mul
add	add	add	add