5 Speicherbelegung für Variablen

Ziel:

Ordne jeder Variablen x statisch, d. h. zur Übersetzungszeit, eine feste (Relativ-)Adresse $\rho\, x$ 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.

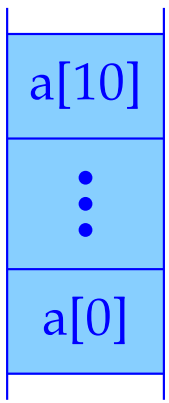
Folglich erhalten wir für die Deklaration $d \equiv t_1\ x_1; \ldots t_k\ x_k;$ (t_i einfach) die Adress-Umgebung ρ mit

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

5.1 Felder

Beispiel: **int** [11] a;

Das Feld a enthält 11 Elemente und benötigt darum 11 Zellen.
$\rho\, a$ 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 $d \equiv t_1\ x_1; \ldots\ t_k\ x_k;$

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

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

Aufgabe:

Erweitere code_L und code_R 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 $\rho\, a + |t| * (\textit{R-Wert von } i)$ ausrechnen. Folglich:

$$
\begin{array}{lll}
\text{code}_L\ a[e]\ \rho & = & \text{loadc}\ (\rho\, a) \\
 & & \text{code}_R\ e\ \rho \\
 & & \text{loadc}\ |t| \\
 & & \text{mul} \\
 & & \text{add}
\end{array}
$$

... oder allgemeiner:

$$
\begin{array}{lcl}
\text{code}_L\ e_1[e_2]\ \rho & = & \text{code}_R\ e_1\ \rho \\
 & & \text{code}_R\ e_2\ \rho \\
 & & \text{loadc}\ |t| \\
 & & \text{mul} \\
 & & \text{add}
\end{array}
$$

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):

$$2[a] \qquad a[2] \qquad a+2$$

5.2 Strukturen

In Modula heißen Strukturen Records.

Vereinfachung:

Komponenten-Namen werden nicht anderweitig verwandt.
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 $t \equiv \textbf{struct}\,\{t_1\ c_1; \ldots t_k\ c_k;\}$. 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:

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

Beispiel:

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

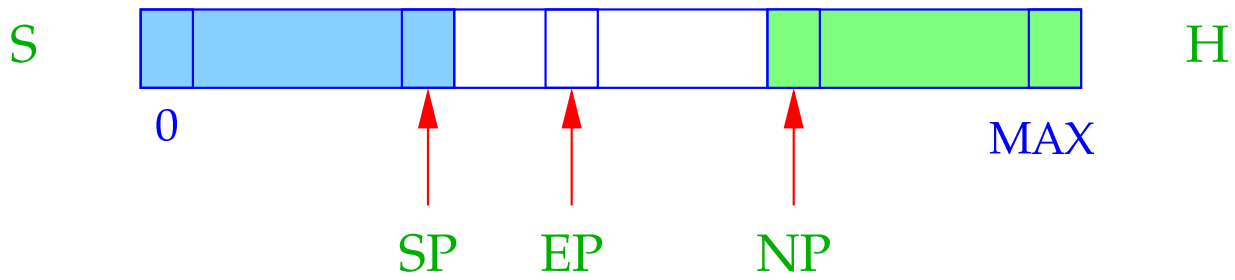
Dann ist

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

6 Zeiger und dynamische Speicherverwaltung

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

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



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

EP $\widehat{=}$ 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 :-(.

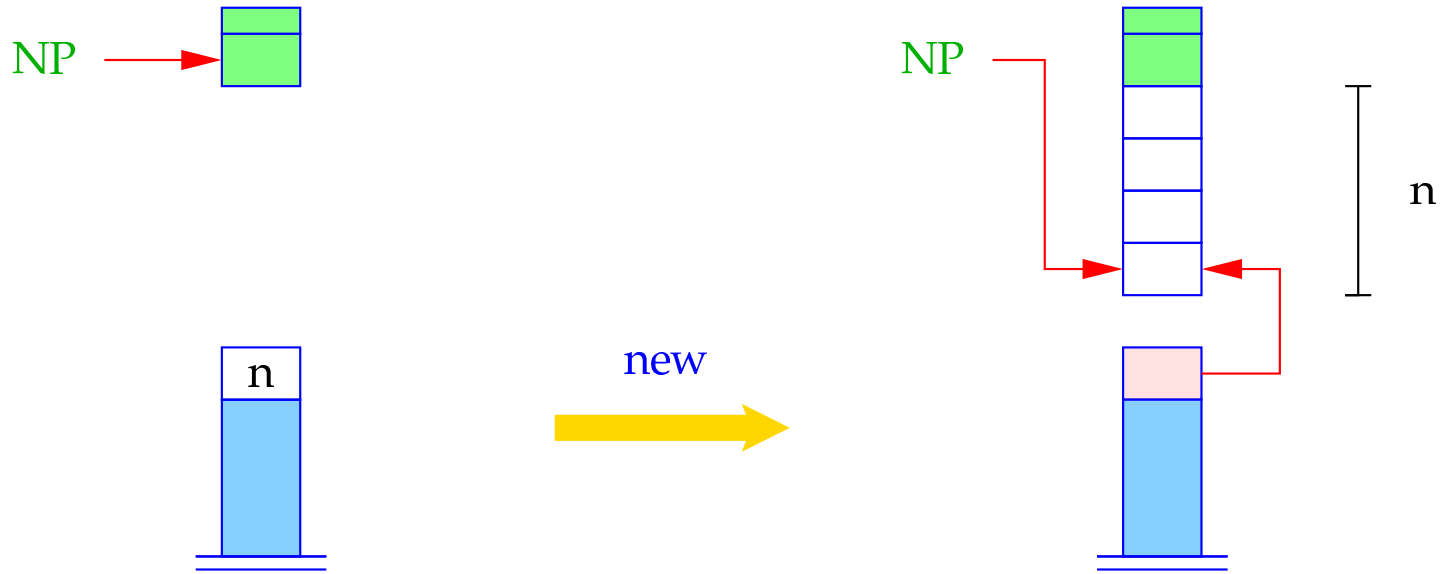
Mit Zeiger (-Werten) rechnen, heißt in der Lage zu sein,

- 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:

$$\begin{aligned} \text{code}_\text{R}\ \textbf{malloc}\,(e)\ \rho \quad &= \quad \text{code}_\text{R}\ e\ \rho \\ & \qquad\ \ \text{new} \end{aligned}$$


NP
n
new
NP
n

```
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 ($\widehat{=}$ L-Wert). Deshalb:

$$\text{code}_\text{R}\ (\&e)\ \rho = \text{code}_\text{L}\ e\ \rho$$

Dereferenzieren von Zeigern:

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

$$\text{code}_\text{L}\ (*e)\ \rho = \text{code}_\text{R}\ e\ \rho$$

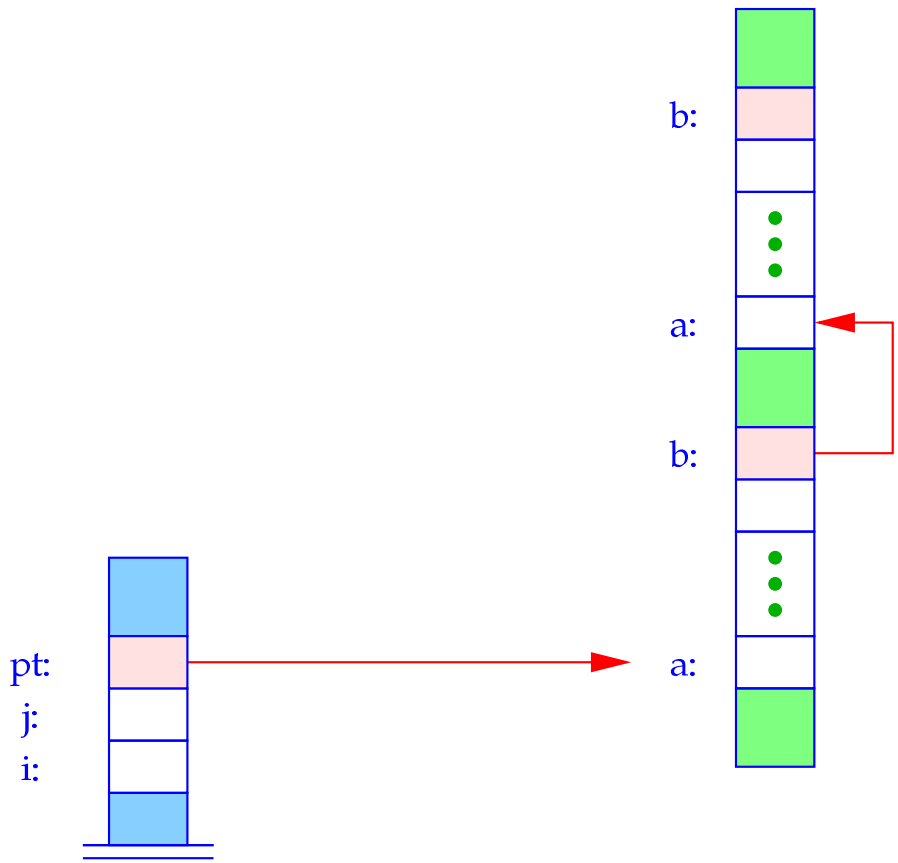
Beispiel: Betrachte für

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

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

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

$$\begin{aligned} \text{code}_L\ (e \to a)\ \rho \quad = \quad & \text{code}_R\ e\ \rho \\ & \text{loadc}\ (\rho\, a) \\ & \text{add} \end{aligned}$$


b:
a:
b:
a:
pt:
j:
i:

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

$\text{code}_L\ e\ \rho$	=	$\text{code}_R\ ((pt \to b) \to a)\ \rho$	=	$\text{code}_R\ ((pt \to b) \to a)\ \rho$
		$\text{code}_R\ (i+1)\ \rho$		loada 1
		loadc 1		loadc 1
		mul		add
		add		loadc 1
				mul
				add

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

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

$$\begin{array}{l}\text{loadc 0}\\ \text{add}\end{array}$$

Damit ergibt sich insgesamt die Folge:

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