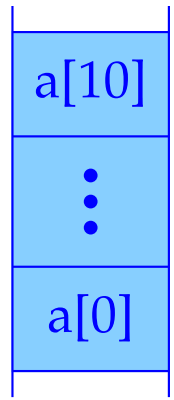
5.1 Arrays

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

The array a consists of 11 components and therefore needs 11 cells.
$\rho\, a$ is the address of the component $a[0]$.



We need a function sizeof (notation: $|\cdot|$), computing the space requirement of a type:

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

Accordingly, we obtain for the declaration $d \equiv t_1\ x_1;\ \ldots\ t_k\ x_k;$

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

Since $|\cdot|$ can be computed at compile time, also ρ can be computed at compile time.

Task:

Extend code_L and code_R to expressions with accesses to array components.

Be $t[c]\ a;$ the declaration of an array a.

To determine the start address of a component $a[i]$, we compute $\rho\, a + |t| * (\textit{R-value of } i)$.

In consequence:

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

... or more general:

$$
\begin{aligned}
\text{code}_\text{L}\ e_1[e_2]\ \rho \quad = \quad & \text{code}_\text{R}\ e_1\ \rho \\
& \text{code}_\text{R}\ e_2\ \rho \\
& \text{loadc}\ |t| \\
& \text{mul} \\
& \text{add}
\end{aligned}
$$

Remark:

- In C, an array is a pointer. A declared array a is a pointer-constant, whose R-value is the start address of the array.
- Formally, we define for an array e: $\text{code}_\text{R}\ e\ \rho = \text{code}_\text{L}\ e\ \rho$
- In C, the following are equivalent (as L-values):

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

Normalization: Array names and expressions evaluating to arrays occur in front of index brackets, index expressions inside the index brackets.

5.2 Structures

In Modula and Pascal, structures are called Records.

Simplification:

Names of structure components are not used elsewhere.
Alternatively, one could manage a separate environment ρ_{st} for each structure type st.

Be **struct** { **int** a; **int** b; } x; part of a declaration list.

- x has as relative address the address of the first cell allocated for the structure.
- The components have addresses relative to the start address of the structure. In the example, these are $a \mapsto 0$, $b \mapsto 1$.

Let $t \equiv \textbf{struct}\,\{t_1\ c_1; \ldots t_k\ c_k;\}$. We have

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

We thus obtain:

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

Example:

Be **struct** { **int** a; **int** b; } x; such that $\rho = \{x \mapsto 13, a \mapsto 0, b \mapsto 1\}$.

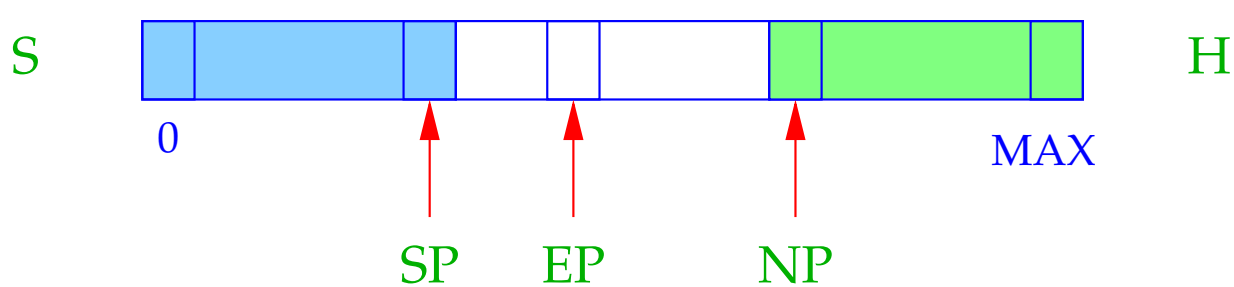
This yields:

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

6 Pointer and Dynamic Storage Management

Pointer allow the access to anonymous, dynamically generated objects, whose life time is not subject to the LIFO-principle.

$\Longrightarrow$ We need another potentially unbounded storage area H – the Heap.



NP $\widehat{=}$ New Pointer; points to the lowest occupied heap cell.

EP $\widehat{=}$ Extreme Pointer; points to the uppermost cell, to which SP can point (during execution of the actual function).

Idea:

- Stack and Heap grow toward each other in S, but must not collide. (Stack Overflow).
- A collision may be caused by an increment of SP or a decrement of NP.
- EP saves us the check for collision at the stack operations.
- The checks at heap allocations are still necessary.

What can we do with pointers (pointer values)?

- set a pointer to a storage cell,
- dereference a pointer, access the value in a storage cell pointed to by a pointer.

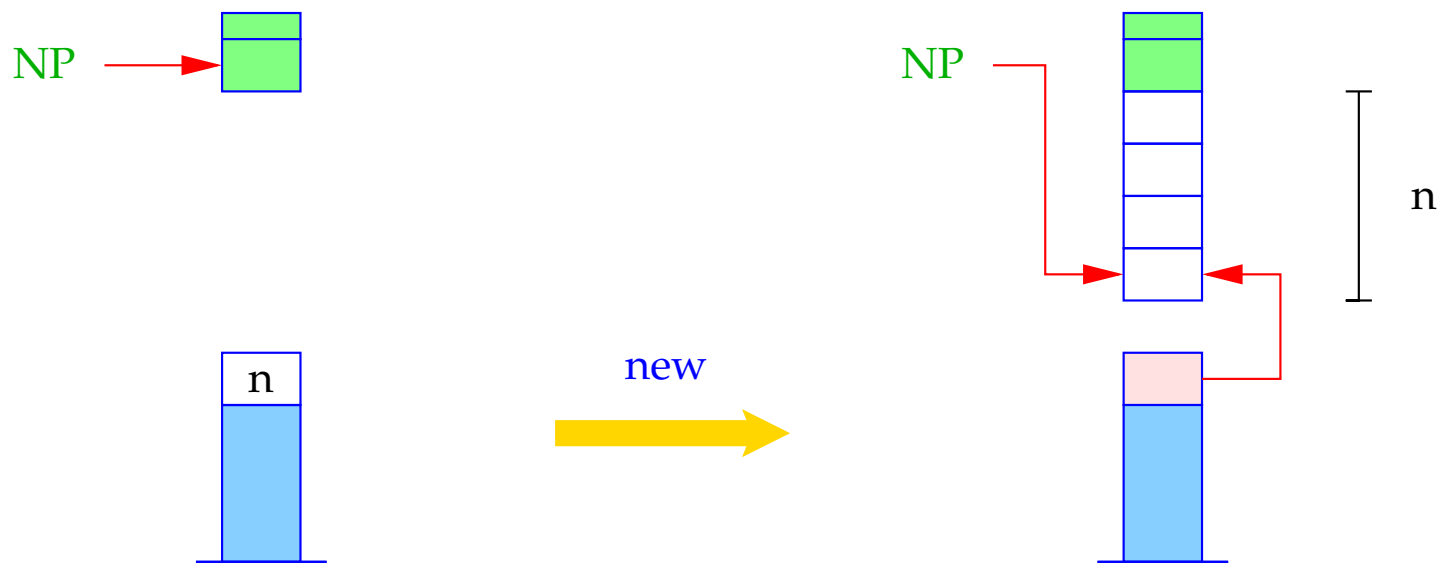
There a two ways to set a pointer:

(1) A call **malloc** (e) reserves a heap area of the size of the value of e and returns a pointer to this area:

$$\text{code}_\text{R}\ \textbf{malloc}\,(e)\ \rho \quad = \quad \text{code}_\text{R}\ e\ \rho$$
$$\text{new}$$

(2) The application of the address operator & to a variable returns a pointer to this variable, i.e. its address ($\widehat{=}$ L-value). Therefore:

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




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

- NULL is a special pointer constant, identified with the integer constant 0.
- In the case of a collision of stack and heap the NULL-pointer is returned.

Dereferencing of Pointers:

The application of the operator $*$ to the expression e returns the contents of the storage cell, whose address is the R-value of e:

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

Example:

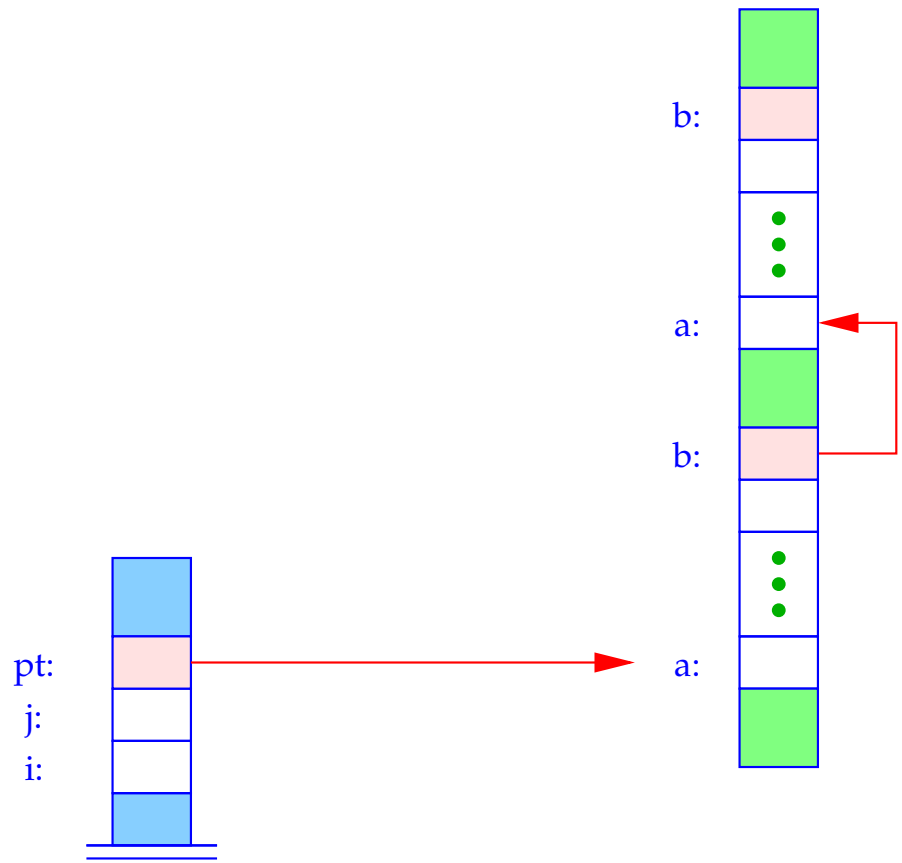
Given the declarations

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

and the expression $((pt \rightarrow b) \rightarrow a)[i+1]$

Because of $e \rightarrow a \equiv (*e).a$ holds:

$$\begin{aligned} \text{code}_\text{L}\ (e \rightarrow a)\ \rho \quad = \quad & \text{code}_\text{R}\ e\ \rho \\ & \text{loadc}\ (\rho\, a) \\ & \text{add} \end{aligned}$$


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

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

$\text{code}_\text{L}\ ((pt \to b) \to a)[i+1]\ \rho$

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

For arrays, their R-value equals their L-value. Therefore:

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

In total, we obtain the instruction sequence:

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

7 Conclusion

We tabulate the cases of the translation of expressions:

$$
\begin{array}{lcll}
\text{code}_{\text{L}}\ (e_1[e_2])\ \rho & = & \text{code}_{\text{R}}\ e_1\ \rho & \\
 & & \text{code}_{\text{R}}\ e_2\ \rho & \\
 & & \text{loadc}\ |t| & \\
 & & \text{mul} & \\
 & & \text{add} & \text{if } e_1 \text{ has type } t* \text{ or } t[] \\
 & & & \\
\text{code}_{\text{L}}\ (e.a)\ \rho & = & \text{code}_{\text{L}}\ e\ \rho & \\
 & & \text{loadc}\ (\rho\, a) & \\
 & & \text{add} &
\end{array}
$$

$$
\begin{aligned}
\text{code}_\text{L}\ (*e)\ \rho &= \text{code}_\text{R}\ e\ \rho \\
\text{code}_\text{L}\ x\ \rho &= \text{loadc}\ (\rho\, x) \\
\text{code}_\text{R}\ (\&e)\ \rho &= \text{code}_\text{L}\ e\ \rho \\
\text{code}_\text{R}\ e\ \rho &= \text{code}_\text{L}\ e\ \rho && \text{if } e \text{ is an array} \\
\text{code}_\text{R}\ (e_1 \,\Box\, e_2)\ \rho &= \text{code}_\text{R}\ e_1\ \rho \\
&\ \text{code}_\text{R}\ e_2\ \rho \\
&\ \text{op} && \text{op instruction for operator '}\Box\text{'}
\end{aligned}
$$

$$
\begin{array}{lcll}
\text{code}_{\text{R}}\ q\ \rho & = & \text{loadc q} & q \text{ constant} \\
\\
\text{code}_{\text{R}}\ (e_1 = e_2)\ \rho & = & \text{code}_{\text{R}}\ e_2\ \rho & \\
 & & \text{code}_{\text{L}}\ e_1\ \rho & \\
 & & \text{store} & \\
\\
\text{code}_{\text{R}}\ e\ \rho & = & \text{code}_{\text{L}}\ e\ \rho & \\
 & & \text{load} & \text{otherwise}
\end{array}
$$

Example: **int** $a[10],\ *b;$ with $\rho = \{a \mapsto 7, b \mapsto 17\}$.

For the statement: $*a = 5;$ we obtain:

$$\begin{array}{lcl}
\text{code}_{\text{L}}\ (*a)\ \rho & = & \text{code}_{\text{R}}\ a\ \rho \quad = \quad \text{code}_{\text{L}}\ a\ \rho \quad = \quad \text{loadc } 7 \\
\text{code}\ (*a = 5;)\ \rho & = & \text{loadc } 5 \\
 & & \text{loadc } 7 \\
 & & \text{store} \\
 & & \text{pop}
\end{array}$$

As an exercise translate:

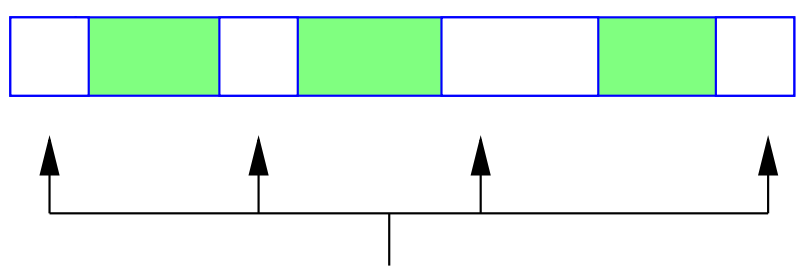
$$s_1 \equiv b = (\&a) + 2; \qquad \text{and} \qquad s_2 \equiv *(b + 3) = 5;$$

$$
\text{code}\ (s_1 s_2)\ \rho \quad = \quad
\begin{array}{lll}
\text{loadc } 7 & & \\
\text{loadc } 2 & & \\
\text{loadc } 10 & // & \textit{size of } \mathbf{int}[10] \\
\text{mul} & // & \textit{scaling} \\
\text{add} & & \\
\text{loadc } 17 & & \\
\text{store} & & \\
\text{pop} & // & \textit{end of } s_1
\end{array}
\qquad
\begin{array}{lll}
\text{loadc } 5 & & \\
\text{loadc } 17 & & \\
\text{load} & & \\
\text{loadc } 3 & & \\
\text{loadc } 1 & // & \textit{size of } \mathbf{int} \\
\text{mul} & // & \textit{scaling} \\
\text{add} & & \\
\text{store} & & \\
\text{pop} & // & \textit{end of } s_2
\end{array}
$$

8 Freeing Occupied Storage

Problems:

- The freed storage area is still referenced by other pointers (dangling references).
- After several deallocations, the storage could look like this (fragmentation):



frei

Potential Solutions:

- Trust the programmer. Manage freed storage in a particular data structure (free list) $\Longrightarrow$ **malloc** or **free** my become expensive.

- Do nothing, i.e.:

$$\text{code } \textbf{free } (e);\ \rho \quad = \quad \text{code}_{\text{R}}\ e\ \rho$$
$$\text{pop}$$

$\Longrightarrow$ simple and (in general) efficient.

- Use an automatic, potentially "conservative" Garbage-Collection, which occasionally collects certainly inaccessible heap space.