5.1 Arrays

Example: \texttt{int [11] a;}

The array \texttt{a} consists of 11 components and therefore needs 11 cells. \texttt{\&a} is the address of the component \texttt{a[0]}.  

\begin{center}
\begin{tabular}{|c|}
\hline
\texttt{a[0]} \\
\hline
\vdots \\
\hline
\texttt{a[10]} \\
\hline
\end{tabular}
\end{center}
We need a function `sizeof` (notation: `| · |`), 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 \( d \equiv t_1 x_1; \ldots t_k x_k; \)

\[
\rho x_1 = 1 \\
\rho x_i = \rho x_{i-1} + |t_{i-1}| \quad \text{for } i > 1
\]

Since `| · |` can be computed at compile time, also \( \rho \) can be computed at compile time.
Task:

Extend \textit{code}_L and \textit{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 \( \rho a + |t| \ast (R\text{-value of } i). \)

In consequence:

\[
\text{code}_L \ a[e] \ \rho \quad = \quad \text{loadc} (\rho a) \\
\text{code}_R \ e \ \rho \\
\text{loadc} \ |t| \\
\text{mul} \\
\text{add}
\]

\( \ldots \) or more general:
\[
\text{code}_L \ e_1[e_2] \ \rho \quad = \quad \text{code}_R \ e_1 \ \rho \\
\text{code}_R \ e_2 \ \rho \\
\text{loadc} \ |t| \\
\text{mul} \\
\text{add}
\]

**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 \( e \): \[ \text{code}_R \ e \ \rho = \text{code}_L \ e \ \rho \]

- In C, the following are equivalent (as L-values): 
  \[ 2[a] \quad a[2] \quad a + 2 \]

**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 $\rho_{st}$ for each structure type $st$.

Be \textbf{struct} \{ \textbf{int} \ a; \textbf{int} \ b; \} \ x; \quad \text{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 $t \equiv \text{struct}\{t_1 \ c_1; \ldots \ t_k \ c_k;\}$. We have

$$|t| = \sum_{i=1}^{k} |t_i|$$

$$\rho c_1 = 0 \quad \text{and}$$

$$\rho c_i = \rho c_{i-1} + |t_{i-1}| \quad \text{for } i > 1$$

We thus obtain:

$$\text{code}_L (e.c) \rho = \text{code}_L e \rho$$

$$\text{loadc } (\rho c)$$

$$\text{add}$$
Example:

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

This yields:

$$\text{code}_L \ (x.b) \ \rho = \text{loadc 13}$$

$$\text{loadc 1}$$

$$\text{add}$$
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.

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

- $NP \equiv \text{New Pointer}$; points to the lowest occupied heap cell.
- $EP \equiv \text{Extreme Pointer}$; 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 are 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:

   \[ \text{code}_R \text{ malloc}(e) \rho = \text{code}_R e \rho \]

   \[ \text{new} \]

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

   \[ \text{code}_R (\&e) \rho = \text{code}_L e \rho \]
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 \( \ast \) to the expression \( e \) returns the contents of the storage cell, whose address is the R-value of \( e \):

\[
\text{code}_L (\ast e) \rho = \text{code}_R e \rho
\]

Example: Given the declarations

```c
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 \( e \rightarrow a \equiv (\ast e).a \) holds:

\[
\text{code}_L (e \rightarrow a) \rho = \text{code}_R e \rho \\
\text{loadc} (\rho a) \\
\text{add}
\]
Be \( \rho = \{ i \mapsto 1, j \mapsto 2, pt \mapsto 3, a \mapsto 0, b \mapsto 7 \} \). Then:

\[
\text{code}_L ((pt \rightarrow b) \rightarrow a)[i + 1] \rho
= \text{code}_R ((pt \rightarrow b) \rightarrow a) \rho
= \text{code}_R ((pt \rightarrow b) \rightarrow a) \rho
\]

\[
\text{code}_R (i + 1) \rho
\]

loada 1
loadc 1
mul
add
add
add loadc 1
mul
add
For arrays, their R-value equals their L-value. Therefore:

\[
\text{code}_R ((pt \rightarrow b) \rightarrow a) \rho = \text{code}_R (pt \rightarrow b) \rho = \begin{align*}
\text{loada} & \ 3 \\
\text{loadc} & \ 0 \\
\text{add} & \ \\
\text{loadc} & \ 7 \\
\text{add} & \\
\text{load} & \\
\text{loadc} & \ 0 \\
\text{add} & \\
\end{align*}
\]

In total, we obtain the instruction sequence:

\[
\begin{align*}
\text{loada} & \ 3 & \text{load} & \ \text{loada} & \ 1 & \text{loadc} & \ 1 \\
\text{loadc} & \ 7 & \text{loadc} & \ 0 & \text{loadc} & \ 1 & \text{mul} \\
\text{add} & \ & \text{add} & \ & \text{add} & \ & \text{add}
\end{align*}
\]
7 Conclusion

We tabulate the cases of the translation of expressions:

\[
\text{code}_L \ (e_1[e_2]) \ \rho \ = \ \text{code}_R \ e_1 \ \rho \\
\text{code}_R \ e_2 \ \rho \\
\text{loadc} \ |t| \\
\text{mul} \\
\text{add} \quad \text{if } e_1 \text{ has type } t* \text{ or } t[]
\]

\[
\text{code}_L \ (e.a) \ \rho \ = \ \text{code}_L \ e \ \rho \\
\text{loadc} \ (\rho \ a) \\
\text{add}
\]
\[ \text{code}_L (\ast e) \rho = \text{code}_R e \rho \]
\[ \text{code}_L x \rho = \text{loadc} (\rho x) \]
\[ \text{code}_R (\& e) \rho = \text{code}_L e \rho \]
\[ \text{code}_R e \rho = \text{code}_L e \rho \quad \text{if } e \text{ is an array} \]
\[ \text{code}_R (e_1 \Box e_2) \rho = \text{code}_R e_1 \rho \]
\[ \quad \text{code}_R e_2 \rho \]
\[ \quad \text{op} \quad \text{op} \text{ instruction for operator ‘} \Box \text{’} \]
\[
\begin{align*}
\text{code}_R \ q \ \rho & \ = \ \text{loadc} \ q \ \quad \text{if } q \ \text{constant} \\
\text{code}_R \ (e_1 = e_2) \ \rho & \ = \ \text{code}_R \ e_2 \ \rho \\
& \quad \text{code}_L \ e_1 \ \rho \\
& \quad \text{store} \\
\text{code}_R \ e \ \rho & \ = \ \text{code}_L \ e \ \rho \\
& \quad \text{load} \ \quad \text{otherwise}
\end{align*}
\]
Example:  \[\text{int } a[10], \ast b; \quad \text{with } \rho = \{a \mapsto 7, b \mapsto 17\}.\]

For the statement:  \[\ast a = 5;\] we obtain:

\[
\begin{align*}
\text{code}_L (\ast a) \rho &= \text{code}_R a \rho = \text{code}_L a \rho = \text{loadc 7} \\
\text{code} (\ast a = 5;) \rho &= \text{loadc 5} \\
&\quad \text{loadc 7} \\
&\quad \text{store} \\
&\quad \text{pop}
\end{align*}
\]

As an exercise translate:

\[s_1 \equiv b = (\&a) + 2;\quad \text{and}\quad s_2 \equiv (b + 3) = 5;\]
code \((s_1s_2)\) \(\rho\) = 

loadc 7
loadc 2
loadc 10 // \textit{size of int}[10]
mul // \textit{scaling}
add
loadc 17
store
pop // \textit{end of s}_1

loadc 5
loadc 17
load
loadc 3
loadc 1 // \textit{size of int}
mul // \textit{scaling}
add
store
pop // \textit{end of s}_2
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):

![Diagram showing fragmented storage with pointers]

frei
Potential Solutions:

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

- Do nothing, i.e.

  \[
  \text{code free}(e); \; \rho \; = \; \text{coder}_e \; \rho \\
  \text{pop}
  \]

  \( \Rightarrow \) simple and (in general) efficient.

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