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**.

\[
\begin{array}{c}
\text{S} & \text{SP} & \text{EP} & \text{NP} & \text{H} \\
0 & & & & \text{MAX}
\end{array}
\]

**NP** \( \equiv \) *New Pointer*: points to the lowest occupied heap cell.

**EP** \( \equiv \) *Extreme Pointer*: points to the uppermost cell, to which \( SP \) can point (during execution of the actual function).
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:

   ```
   code_ malloc (e) \rho = code_ e \rho
   ```

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

   ```
   code_ (\&e) \rho = code_ e \rho
   ```

**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}_1 \text{(} e \text{)} \rho = \text{code}_0 \text{ e } \rho
```

**Example:**

Given the declarations:

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

and the expression `((pt -> b) -> a)[i+1]`

Because of `e \rightarrow a = (e).a` holds:

```
\text{code}_1 \text{(} e \rightarrow a \text{)} \rho = \text{code}_0 \text{ e } \rho
```

```
\text{loadc} (\rho a)
```

```
\text{add}
```
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 \( e \):

\[
\text{code}_{\text{L}} \ (\ast e) \rho = \text{code}_{\text{R}} e \rho
\]

Example:

Given the declarations

\[
\begin{align*}
\text{struct } t \{ \text{int } i[7]; \text{struct } t * b; j; \} \\
\text{int } i, j;
\end{align*}
\]

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

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

\[
\text{code}_{\text{L}} \ (e \rightarrow a) \rho = \text{code}_{\text{R}} e \rho
\]

\[
\begin{align*}
\text{loadc} \ (\rho a) \\
\text{add}
\end{align*}
\]
Dereferencing of Pointers:

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

$$\text{decode}_e (\star e) \rho = \text{decode}_e e \rho$$

**Example:** Given the declarations

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

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

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

$$\text{decode}_e (e \rightarrow a) \rho = \text{decode}_e e \rho \leftarrow \text{loadc} (\rho a) \leftarrow \text{add}$$

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

$$\text{decode}_e \left( (\star (pt \rightarrow b) \rightarrow a)[i + 1] \rho \right) = \text{decode}_e \left( (\star (pt \rightarrow b) \rightarrow a) \rho \right)$$

$$= \text{loadc} 1 \leftarrow \text{loadc} 1 \leftarrow \text{add} \leftarrow \text{mul} \leftarrow \text{add}$$
For arrays, their R-value equals their L-value. Therefore:

\[
\text{code}_{\text{e}} ((\mathit{pt} \to \mathit{b}) \to \mathit{a}) \rho = \text{code}_{\text{e}} (\mathit{pt} \to \mathit{b}) \rho = \text{load}_{\mathit{a}} \rho
\]

In total, we obtain the instruction sequence:

<table>
<thead>
<tr>
<th>loada 3</th>
<th>load</th>
<th>loadc 1</th>
<th>loadc 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>loadc 7</td>
<td>loadc 0</td>
<td>loadc 1</td>
<td>mul</td>
</tr>
<tr>
<td>add</td>
<td>add</td>
<td>add</td>
<td>add</td>
</tr>
</tbody>
</table>

\[
\text{code}_{\mathit{e}} (\ast e) \rho = \text{code}_{\mathit{e}} e \rho
\]

\[
\text{code}_{\mathit{e}} x \rho = \text{loadc} (\rho x)
\]

\[
\text{code}_{\mathit{e}} (\& e) \rho = \text{code}_{\mathit{e}} e \rho
\]

\[
\text{code}_{\mathit{e}} e \rho = \text{code}_{\mathit{e}} e \rho \quad \text{if } e \text{ is an array}
\]

\[
\text{code}_{\mathit{e}} (e_1 \sqcap e_2) \rho = \text{code}_{\mathit{e}} e_1 \rho \quad \text{op} \quad \text{code}_{\mathit{e}} e_2 \rho
\]

\[
\begin{align*}
\text{code}_{\mathit{q}} q \rho & = \text{loadc} q \rho \quad q \text{ constant} \\
\text{code}_{\mathit{q}} (e_1 = e_2) \rho & = \text{code}_{\mathit{e}} e_2 \rho \quad \text{code}_{\mathit{e}} e_1 \rho \\
\text{code}_{\mathit{e}} e \rho & = \text{load} e \rho \quad \text{store} \\
\text{code}_{\mathit{e}} e \rho & = \text{code}_{\mathit{e}} e \rho \quad \text{op instruction for operator ‘\&’}
\end{align*}
\]
Example: \[ \text{int } a[10], (b)[10]; \text{ with } \rho = \{ a \mapsto 7, b \mapsto 17 \}. \]

For the statement: \[ *a = 5; \]

we obtain:

\[
\begin{align*}
\text{code}_{L}_{\rho} (\star a) \rho &= \text{code}_{L} \, a \rho = \text{code}_{L} \, a \rho = \text{load} \, 7 \\
\text{code}_{L} (\star a = 5) \rho &= \text{load} \, 5 \\
& \quad \text{store} \\
& \quad \text{pop}
\end{align*}
\]

As an exercise translate:

\[
\begin{align*}
s_1 &= b = (\&a) + 2; \quad \text{and} \quad s_2 = (b + 3)[10] = 5;
\end{align*}
\]

\[
\begin{align*}
\text{code } (s_1, s_2) \rho &= \text{load} \, 7 \\
& \quad \text{load} \, 2 \\
& \quad \text{load} \, 10 \quad \text{// size of int[10]} \\
& \quad \text{load} \\
& \quad \text{mul} \quad \text{// scaling} \\
& \quad \text{load} \, 3 \\
& \quad \text{add} \\
& \quad \text{load} \, 10 \quad \text{// size of int[10]} \\
& \quad \text{load} \\
& \quad \text{mul} \quad \text{// scaling} \\
& \quad \text{add} \\
& \quad \text{load} \\
& \quad \text{store} \\
& \quad \text{pop} \quad \text{// end of } s_1 \\
& \quad \text{store} \\
& \quad \text{pop} \quad \text{// end of } s_2
\end{align*}
\]

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) >>> malloc or free my become expensive.
- Do nothing, i.e.: 
  \[ \text{code free } (\epsilon); \rho = \text{codeg } \epsilon \rho \]
  \[ \text{pop} \]
  \[ \implies \text{simple and (in general) efficient.} \]
- Use an automatic, potentially “conservative” Garbage-Collection, which occasionally collects certainly inaccessible heap space.

9 Functions

The definition of a function consists of:
- a name by which it can be called;
- a specification of the formal parameters;
- a possible result type;
- a block of statements.
In C, we have:
  \[ \text{codeg } f ; \rho = \text{load } c : f \]
  \[ \implies \text{start address of the code for } f \]
  \[ \implies \text{Function names must be maintained within the address environment!} \]

Potential Solutions:
- Trust the programmer. Manage freed storage in a particular data structure (free list) >>> malloc or free my become expensive.
- Do nothing, i.e.: 
  \[ \text{code free } (\epsilon); \rho = \text{codeg } \epsilon \rho \]
  \[ \text{pop} \]
  \[ \implies \text{simple and (in general) efficient.} \]
- Use an automatic, potentially “conservative” Garbage-Collection, which occasionally collects certainly inaccessible heap space.

Example

```c
int fac (int x) {
    if (x <= 0) return 1;
    else return x * fac (x - 1);
}
```

main () {
    int n = fac (2) + fac (1);
    printf ("%d", n);
}

At every point of execution, several instances (calls) of the same function may be active, i.e., have been started, but not yet completed.
The recursion tree of the example:
We conclude:

The **formal parameters** and **local variables** of the different calls of the same function (the instances) must be kept separate.

**Idea**

Allocate a dedicated memory block for each call of a function. In sequential programming languages, these memory blocks may be maintained on a stack. Therefore, they are also called **stack frames**.

---

**Caveat**

- The local variables receive relative addresses \(+1, +2\).
- The formal parameters are placed below the organizational cells and therefore have **negative** addresses relative to \(FP\).
- This organization is particularly well suited for function calls with variable number of arguments as, e.g., for `printf`.
- The memory block of parameters is recycled for storing the return value of the function \(\Rightarrow\).

**Simplification**

The return value fits into a single memory cell.

---

**9.1 Memory Organization for Functions**

\(f(x_1, x_2)\)

**FP** ▶ Frame Pointer; points to the last organizational cell and is used for addressing the formal parameters and local variables.
Caveat

- The local variables receive relative addresses ++1, ++2, ... 
- The formal parameters are placed below the organizational cells and therefore have negative addresses relative to FP :-(
- This organization is particularly well suited for function calls with variable number of arguments as, e.g., for printf.
- The memory block of parameters is recycled for storing the return value of the function :-(

Simplification: The return value fits into a single memory cell.

9.1 Memory Organization for Functions

| SP | lokale Variablen |
| FP | organisatorische Zellen |
|    | formale Parameter / Funktionswert |

FP = Frame Pointer; points to the last organizational cell and is used for addressing the formal parameters and local variables.
9.2 Determining Address Environments

We distinguish two kinds of variables:
1. **global/extern** that are defined outside of functions;
2. **local/extern/automatic** (including formal parameters) which are defined inside functions.

The address environment $\rho$ maps names onto pairs $\langle \text{tag}, \alpha \rangle \in \{G, L\} \times \mathbb{Z}$.

**Caveat**
- In general, there are further refined grades of visibility of variables.
- Different parts of a program may be translated relative to different address environments!
Address Environments Occurring in the Program:

0 Out of the Function Definitions:

\[ \rho_0 : \]
\[ i \rightarrow (G_1, 1) \]
\[ l \rightarrow (G_1, 2) \]
\[ \text{ith} \rightarrow (G_{\text{ith}}, \text{ith}) \]
\[ \text{main} \rightarrow (G_{\text{main}}, \text{main}) \]
...

1 Inside of ith:

\[ \rho_1 : \]
\[ i \rightarrow (L_i, -4) \]
\[ x \rightarrow (L_i, -3) \]
\[ l \rightarrow (G_2, 2) \]
\[ \text{ith} \rightarrow (G_{\text{ith}}, \text{ith}) \]
\[ \text{main} \rightarrow (G_{\text{main}}, \text{main}) \]
...

%}

Address Environments Occurring in the Program:

0 Out of the Function Definitions:

\[ \rho_0 : \]
\[ i \rightarrow (G_1, 1) \]
\[ l \rightarrow (G_2, 2) \]
\[ \text{ith} \rightarrow (G_{\text{ith}}, \text{ith}) \]
\[ \text{main} \rightarrow (G_{\text{main}}, \text{main}) \]
...

1 Inside of ith:

\[ \rho_1 : \]
\[ i \rightarrow (L_i, -4) \]
\[ x \rightarrow (L_i, -3) \]
\[ l \rightarrow (G_2, 2) \]
\[ \text{ith} \rightarrow (G_{\text{ith}}, \text{ith}) \]
\[ \text{main} \rightarrow (G_{\text{main}}, \text{main}) \]
...

%}

Address Environments Occurring in the Program:

0 Out of the Function Definitions:

\[ \rho_0 : \]
\[ i \rightarrow (G_1, 1) \]
\[ l \rightarrow (G_2, 2) \]
\[ \text{ith} \rightarrow (G_{\text{ith}}, \text{ith}) \]
\[ \text{main} \rightarrow (G_{\text{main}}, \text{main}) \]
...

1 Inside of ith:

\[ \rho_1 : \]
\[ i \rightarrow (L_i, -4) \]
\[ x \rightarrow (L_i, -3) \]
\[ l \rightarrow (G_2, 2) \]
\[ \text{ith} \rightarrow (G_{\text{ith}}, \text{ith}) \]
\[ \text{main} \rightarrow (G_{\text{main}}, \text{main}) \]
...

%}

Example

0 int i;

struct list {
    int info;
    struct list * next;
} * l;

scanf("%d", &i);
scansion(&l);

1 int ith (struct list * x, int i) {
    if (i <= 1) return x ->info;
    else return ith (x ->next, i - 1);
}

2 main () {
    int k;
    printf("%d\n", ith (l, i));
}

%
9.1 Memory Organization for Functions

SP → PCode

FP → PCold
FPold
EPold

lokale Variablen
organisatorische Zellen
formale Parameter / Funktionswert

FP ≡ Frame Pointer; points to the last organizational cell and is used for addressing the formal parameters and local variables.

Address Environments Occurring in the Program:

- **Outside of the Function Definitions:**
  - \( \rho_0 : i \rightarrow (G, 1) \)
  - \( l \rightarrow (G, 2) \)
  - \( \text{ith} \rightarrow (G_{\_ith}) \)
  - \( \text{main} \rightarrow (G_{\_main}) \)
  - ...

- **Inside of ith:**
  - \( \rho_1 : i \rightarrow (L, -4) \)
  - \( x \rightarrow (L, -3) \)
  - \( l \rightarrow (G, 2) \)
  - \( \text{ith} \rightarrow (G_{\_ith}) \)
  - \( \text{main} \rightarrow (G_{\_main}) \)
  - ...

**Caveat**
- The actual parameters are evaluated from right to left !!
- The first parameter resides directly below the organizational cells :)
- For a prototype \( \tau f(x_1, \ldots, x_n) \) we define:
  - \( x_1 \rightarrow (L, -2 - |x_1|) \)
  - \( x_2 \rightarrow (L, -2 - |x_1| - \ldots - |x_n|) \)

**Caveat**
- The actual parameters are evaluated from right to left !!
- The first parameter resides directly below the organizational cells :)
- For a prototype \( \tau f(x_1, x_2, \ldots, x_n) \) we define:
  - \( x_1 \rightarrow (L, -2 - |x_1|) \)
  - \( x_2 \rightarrow (L, -2 - |x_1| - \ldots - |x_n|) \)

**Inside of main:**
- \( \rho_2 : i \rightarrow (G, 1) \)
- \( l \rightarrow (G, 2) \)
- \( k \rightarrow (L, 1) \)
- \( \text{ith} \rightarrow (G_{\_ith}) \)
- \( \text{main} \rightarrow (G_{\_main}) \)
- ...

FP + 1
9.3 Calling/Entering and Exiting/Leaving Functions

Assume that \( f \) is the current function, i.e., the caller, and \( f \) calls the function \( g \), i.e., the callee.

The code for the call must be distributed between the caller and the callee.
The distribution can only be such that the code depending on information of the caller must be generated for the caller and likewise for the callee.

Caveat
The space requirements of the actual parameters is only known to the caller.

Actions when entering \( g \):
1. Evaluating the actual parameters, \( \{ \text{mark}\} \)
2. Saving of FP, EP, \( \{ \text{call}\} \)
3. Determining the start address of \( g \)
4. Setting of the new FP, \( \{ \text{call}\} \)
5. Saving PC and jump to the beginning of \( g \)
6. Setting of new EP, \( \{ \text{enter}\} \)
7. Allocating of local variables, \( \{ \text{alloc}\} \)

Accordingly, we obtain for a call to a function with at least one parameter and one return value:

\[
\text{code} g(e_1, \ldots, e_n) \rho = \text{code} e_n \rho
\]

\[
\vdots
\]

\[
\text{code} e_1 \rho
\]

\[
\text{mark}
\]

\[
\text{code} g \rho
\]

\[
\text{call}
\]

\[
\text{slide} (m - 1)
\]

where \( m \) is the size of the actual parameters.
Remark

- Of every expression which is passed as a parameter, we determine the R-value \( \text{call-by-value} \) passing of parameters.
- The function \( g \) may as well be denoted by an expression, dessen \text{R-Wert} die Anfangs-Adresse der aufzurufenden Funktion liefert ...

\[
\begin{align*}
\text{code}_e f \, \rho &= \text{loadc} (\rho f) & f \text{ name of a function} \\
\text{code}_e (e \cdot) \rho &= \text{codec} e \rho & e \text{ function pointer} \\
\text{codec} e \rho &= \text{codec}_e e \rho & e \text{ a structure of size } k \\
\text{move k} & &
\end{align*}
\]

where

- Similar to declared arrays, function names are interpreted as constant \text{pointers} onto function code. Thus, the R-value of this pointer is the start address of the function.
- 
  \text{Caveat!} For a variable \( \text{int} \ (\cdot) \ g \), the two calls

  \[
  \begin{array}{ll}
  (g)() \\
  g()
  \end{array}
  \]

  are equivalent! By means of normalization, the dereferencing of function pointers can be considered as redundant \( \therefore \)
- During passing of parameters, these are copied.

Consequently,

\[
\begin{align*}
\text{codec}_e f \, \rho &= \text{loadc} (\rho f) & f \text{ name of a function} \\
\text{codec}_e (e \cdot) \rho &= \text{codec} e \rho & e \text{ function pointer} \\
\text{codec} e \rho &= \text{codec}_e e \rho & e \text{ a structure of size } k \\
\text{move k} & &
\end{align*}
\]

where
• Similar to declared arrays, function names are interpreted as constant
    points onto function code. Thus, the R-value of this pointer is the start
    address of the function.
• Caveat! For a variable \texttt{int ( * ) g;} the two calls
    \[(g)() \quad \text{and} \quad g()\]
    are equivalent! By means of \texttt{normalization}, the dereferencing of function
    pointers can be considered as redundant \texttt{:-)}
• During passing of parameters, these are copied.

Consequently,

\[
\begin{align*}
\text{code}_{e} f \rho & = \text{loadc} ( f \rho ) & f \text{ name of a function} \\
\text{code}_{e} ( * e ) \rho & = \text{codec} e \rho & e \text{ function pointer} \\
\text{code}_{e} e \rho & = \text{codec} e \rho & e \text{ a structure of size } k
\end{align*}
\]

where

The instruction \texttt{mark} saves the registers \texttt{FP} and \texttt{EP} onto the stack:

\[
\begin{align*}
\text{FP} & \rightarrow \text{SP} + 1 \rightarrow \text{EP} \\
\text{S}[\text{SP} + 1] & = \text{EP} \\
\text{S}[\text{SP} + 2] & = \text{FP} \\
\text{SP} & = \text{SP} + 2
\end{align*}
\]

The instruction \texttt{call} saves the return address and sets \texttt{FP} and \texttt{PC} onto the new values:

\[
\begin{align*}
\text{FP} & \rightarrow \text{SP} + 1 \\
\text{PC} & \rightarrow \text{SP} + 2
\end{align*}
\]

The instruction \texttt{slide} copies the return values into the correct memory cell:

\[
\begin{align*}
tmp & = \text{S}[\text{SP}] \\
\text{S}[\text{SP}] & = \text{PC} \\
\text{FP} & = \text{SP} \\
\text{PC} & = \text{tmp}
\end{align*}
\]

\[
\begin{align*}
tmp & = \text{S}[\text{SP}] \\
\text{SP} & = \text{SP} - m \\
\text{S}[\text{SP}] & = \text{tmp}
\end{align*}
\]
Accordingly, we translate a function definition:

```
\text{code } f \{ (\text{spec}) \{ V_{\text{defs}} \} \}_{\rho} = \\
\begin{array}{l}
  \text{\_f: enter } q \quad \text{// initialize EP} \\
  \text{\_alloc } k \quad \text{// allocate the local variables} \\
  \text{\_code } s_{\rho} \\
  \text{\_return } \text{// return from call}
\end{array}
```

where 
\[
q = \text{max} + k \\
\text{max} = \text{maximal length of the local stack} \\
k = \text{size of the local variables} \\
\rho_f = \text{address environment for } f \\
\text{// takes spec, V_{\text{defs}} and } \rho \text{ into account}
\]

The instruction `enter q` sets the EP to the new value. If not enough space is available, program execution terminates.

The instruction `alloc k` allocates memory for locals on the stack.

```
SP = SP + k;
```

The instruction `return` pops the current stack frame. This means it restores the registers PC, EP and FP and returns the return value on top of the stack.

```
PC = S[FP]; EP = S[FP-2]; \\
\text{if } (EP \geq NP) \text{ Error ("Stack Overflow");} \\
SP = FP-3; FP = S[SP+2];
```
The instruction `return` pops the current stack frame. This means it restores the registers PC, EP and FP and returns the return value on top of the stack.

PC = S[FP]; EP = S[FP-2];
if (EP ≥ NP) Error ("Stack Overflow");
SP = FP-3; FP = S[SP+2];

---

9.4 Access to Variables, Formal Parameters and Returning of Values

Accesses to local variables or formal parameters are relative to the current FP. Accordingly, we modify `coder` for names of variables.

For \( \rho \cdot x = (\text{tag}, \ell) \), we define

\[
\text{coder}_1 \cdot x \cdot \rho = \begin{cases} 
\text{loadc}_j & \text{tag} = G \\
\text{loadc}_j & \text{tag} = l 
\end{cases}
\]

---

The instruction `loadc` \( j \) computes the sum of \( \text{FP} \) and \( j \).

FP \( f \)

loadc \( j \)

FP \( f+j \)

SP++;
S[SP] = FP+j;

---

As an optimization, we introduce analogously to `loada` \( j \) and `stora` \( j \) the new instructions `loadr` \( j \) and `storer` \( j \):

\[
\text{loadr}_j = \frac{\text{loadc}_j}{\text{load}}
\]

\[
\text{storer}_j = \frac{\text{loadc}_j}{\text{store}}
\]
The code for \( \text{return } c; \) corresponds to an assignment to a variable with relative address \(-3\).

\[
\text{code } \begin{cases}
\text{return } c; \quad \rho = \text{code } c; \\
\text{storer } -3 \quad \text{return }
\end{cases}
\]

Example For function

\[
\text{int fac (int x) }
\begin{cases}
\text{if } (x \leq 0) \text{return 1; } \\
\text{else return } x = \text{fac } (x - 1); \\
\end{cases}
\]

we generate:

As an optimization, we introduce analogously to \( \text{loada } j \) and \( \text{storea } j \) the new instructions \( \text{loadr } j \) and \( \text{storer } j \):

\[
\begin{align*}
\text{loadr } j &= \text{loadrc } j \\
\text{storer } j &= \text{loadrc } j;
\end{align*}
\]