Dereferencing of Pointers:

The application of the operator \( * \) to the expression \( e \) returns the contents of the storage cell, whose address is the K-value of \( e \):

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

Example  Given the declarations

\[
\text{struct}\ t\ \{\ \text{int}\ a[7];\ \text{struct}\ t + pt;\ \}
\]

\text{int}\ i, j;

\text{struct}\ t + pt;

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

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

\[
\text{code}_L\ (e \rightarrow a)\ \rho = \text{code}_E\ e\ \rho
\]

\[
\begin{array}{l}
\text{loadc}\ (\rho a) \\
\text{add}
\end{array}
\]

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

\[
\begin{align*}
\text{code}_L\ ((pt \rightarrow b) \rightarrow a)[i + 1] \rho \\
\ = \ & \text{code}_E\ ((pt \rightarrow b) \rightarrow a)\ \rho \\
\ & \text{code}_E\ (i + 1)\ \rho \\
\ & \text{loadc}\ 1 \\
\ & \text{add}
\end{align*}
\]

\[
\begin{align*}
\text{loadc}\ 1 \\
\text{add}
\end{align*}
\]
For arrays, their R-value equals their L-value. Therefore:

\[
\text{code}_\text{L}((p \to b) \to a) \, \rho = \text{code}_\text{L}(pt \to b) \, \rho = \text{load\_a}  \\
\text{load\_0}  \quad \text{add}  \\
\text{load\_7}  \quad \text{add}  \\
\text{load\_0}  \quad \text{add}  \\
\text{load\_1}  \quad \text{add}  \\
\text{mul} \quad \text{add} \quad \text{if } e_1 \text{ has type } ts \text{ or } t[]
\]

In total, we obtain the instruction sequence:

\[
\text{load\_3}  \\
\text{load}  \\
\text{load\_1}  \\
\text{load\_1}  \\
\text{load\_1}  \\
\text{load\_0}  \\
\text{add}  \\
\text{add}  \\
\text{add}  \\
\text{add}
\]

7 Conclusion

We tabulate the cases of the translation of expressions:

\[
\text{code}_\text{L}([e_1][e_2]) \, \rho = \text{code}_\text{L} e_1 \, \rho  \\
\text{code}_\text{R} e_2 \, \rho  \\
\text{load} \, [t]  \\
\text{mul}  \\
\text{add}  \\
\text{if } e_1 \text{ has type } ts \text{ or } t[]
\]

\[
\text{code}_\text{L}(e, a) \, \rho = \text{code}_\text{L} e \, \rho  \\
\text{load} \, (p\, a)  \\
\text{add}
\]

\[
\text{code}_\text{R} q \, \rho = \text{load} \, q  \\
\text{q constant}
\]

\[
\text{code}_\text{R} (e_1 = e_2) \, \rho = \text{code}_\text{R} e_2 \, \rho  \\
\text{code}_\text{L} e_1 \, \rho  \\
\text{store}
\]

\[
\text{code}_\text{R} e \, \rho = \text{code}_\text{L} e \, \rho  \\
\text{load}  \\
\text{otherwise}
\]

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

\[
\text{code}_\text{L} x \, \rho = \text{load} \, (p\, x)
\]

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

\[
\text{code}_\text{L} e \, \rho = \text{code}_\text{L} e \, \rho  \\
\text{if } e \text{ is an array}
\]

\[
\text{code}_\text{L} (e_1 \sqcap e_2) \, \rho = \text{code}_\text{L} e_1 \, \rho  \\
\text{code}_\text{L} e_2 \, \rho  \\
\text{op}  \\
\text{op instruction for operator } \sqcap'
\]
Example  int  a[10], (+b)[10];  with  \( \rho = \{ a \mapsto 7, b \mapsto 17 \} \).

For the statement:  \( +a = 5; \)  we obtain:

\[
\text{code}_L(\ast a) \ \rho = \text{code}_R a \ \rho = \text{code}_L a \ \rho = \text{load}_7
\]

\[
\text{load}_5
\]

\[
\text{load}_7 \quad \text{store}
\]

\[
\text{pop}
\]

As an exercise translate:

\[
s_1 \equiv b = (\&a) + 2; \quad \text{and} \quad s_2 \equiv \ast(b + 3)[0] = 5;
\]
\[\text{Example} \quad \text{int } a[10], \,(+b)[10]; \quad \text{with } \rho = \{a \mapsto 7, b \mapsto 17\}.\]

For the statement: \(+a = 5;\) we obtain:

\[
\begin{align*}
\text{code}_1 \,(+x) \rho &= \text{code}_2 \,a \rho = \text{code}_2 \,a \rho = \text{loadc} 7 \\
\text{code} \,(+a = 5) \rho &= \text{loadc} 7 \\
\end{align*}
\]

As an exercise translate:

\[s_1 = b = (+a) + 2; \quad \text{and} \quad s_2 = (+b + 3)[0] = 5;\]

8 Freeing Occupied Storage

Problems:
\begin{itemize}
  \item The freed storage area is still referenced by other pointers (dangling references).
  \item After several deallocations, the storage could look like this (fragmentation):
\end{itemize}
Potential Solutions:

- Trust the programmer. Manage freed storage in a particular data structure (free list) \(\implies\) malloc or free my become expensive.
- Do nothing, i.e.,
  
  \[
  \begin{align*}
  \text{code} \ & \ free \ (e) ; \ \rho \ & \ = \ & \ \text{code}\_\text{ex} \ e \ \rho \\
  \ & \ \text{pop}
  \end{align*}
  \]

\(\implies\) 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:

\[
\begin{align*}
\text{code}_R \ f \ & \ = \ & \ \text{loadc}_R \ f \\
\end{align*}
\]

\(\implies\) start address of the code for \(f\)

\(\implies\) Function names must be maintained within the address environment!

---

Example

```c
int fac (int n) {
  if (x \leq 0) return 1;
  else return x \cdot fac (x - 1);
}
main () {
  int n;
  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 \(+1, +2, \ldots\).
- The formal parameters are placed below the organizational cells and therefore have negative addresses relative to FP \(\rightarrow\).
- 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**

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 \( +1, +2, \ldots \).
- The formal parameters are placed below the organizational cells and therefore have negative addresses relative to FP \( \Rightarrow \)
- This organization is particularly well suited for function calls with variable number of arguments as, e.g., for \texttt{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

\[
\begin{align*}
\text{SP} & \quad \text{local variables} \\
\text{FP} & \quad \text{organizational cells} \\
& \quad \text{formal parameters / return value} \\
\end{align*}
\]

\(\text{FP} = \text{Frame Pointer};\) points to the last organizational cell and is used for addressing the formal parameters and local variables.
9.1 Memory Organization for Functions

![Diagram of memory organization for functions]

- **SP**
  - Local variables

- **FP**
  - PCold
  - FPold
  - EPold
  - Formal parameters / return value

**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/intern/automatic (including formal parameters) which are defined inside functions.

The address environment $\rho$ maps names onto pairs $(\text{tag}, a) \in \{G, L\} \times \mathbb{Z}$.

**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!

---

**Example**

```c
0 int i;
struct list {
    int info;
    struct list * next;
} = i;

2 main () {
    int k;
    scanf("%d", &i);
    scanlist (&k);
    printf("\n%d\n", ith (k));
}
```

```c
1 int ith (struct list * x, int i) {
    if (i < 1) return x->info;
    else return ith (x->next, i - 1);
}
```
Address Environments Occurring in the Program:

0 Before the Function Definitions:

\[ \rho_0 : \]
\[ i \mapsto (G, 1) \]
\[ l \mapsto (G, 2) \]
\[ \ldots \]

1 Inside of \texttt{ith}:

\[ \rho_1 : \]
\[ i \mapsto (L, -4) \]
\[ x \mapsto (L, -3) \]
\[ l \mapsto (G, 2) \]
\[ \texttt{ith} \mapsto (G, \_\texttt{ith}) \]
\[ \ldots \]