What can we do with pointers (pointer values)?

- **set** a pointer to a storage cell,
- **derefence** 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:

   ```
   codeR malloc(e) p = codeR e p
   new
   ```

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

   ```
   codeR (&e) p = codeL e p
   ```

---

What can we do with pointers (pointer values)?

- **set** a pointer to a storage cell,
- **derefence** 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:

   ```
   codeR malloc(e) p = codeR e p
   new
   ```

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

   ```
   codeR (&e) p = codeL e p
   ```

---

- **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`:

```
codeL (*e) ρ = codeL e ρ
```

**Example**  Given the declarations

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

and the expression `(*(pt + b))[i + 1]`

Because of `e → a ≡ (*e).a` holds:

```
codeL (e → a) ρ = codeL e ρ
```

```
load ρa
add
```

57

58

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`:

```
codeL (*e) ρ = codeL e ρ
```

**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 → a ≡ (*e).a` holds:

```
codeL (e → a) ρ = codeL e ρ
```

```
load ρa
add
```

57

58

Be  `ρ = {i ↦ 1, j ↦ 2, pt ↦ 3, a ↦ 0, b ↦ 7}`. Then:

```
codeL (((pt → b) → a)[i + 1]) ρ = codeL (((pt → b) → a) ρ
```

```
codeL (i + 1) ρ
load 1
load 1
mul
add
```

```
codeL (pt → b) ρ
load 1
add
mul
add
```

59
For arrays, their R-value equals their L-value. Therefore:

\[
\text{code}_\text{R}((pt \to b) \to a) \; \rho = \text{code}_\text{R}(pt \to b) \; \rho = \text{loada} \; 3 \\
\text{loadc} \; 0 \\
\text{add} \\
\text{load} \\
\text{loadc} \; 0 \\
\text{add}
\]

In total, we obtain the instruction sequence:

\[
\text{loada} \; 3 \\
\text{load} \\
\text{loada} \; 1 \\
\text{loadc} \; 7 \\
\text{loadc} \; 0 \\
\text{add} \\
\text{add} \\
\text{add} \\
\text{add}
\]

7 Conclusion

We tabulate the cases of the translation of expressions:

\[
\text{code}_\text{L}((e_1[e_2]) \; \rho = \text{code}_\text{R} \; e_1 \; \rho \\
\text{code}_\text{R} \; e_2 \; \rho \\
\text{loadc} \; [t] \\
\text{mul} \\
\text{add} \\
\text{if } e_1 \text{ has type } t\ast \text{ or } t[]
\]

\[
\text{code}_\text{L}((e \ast a) \; \rho = \text{code}_\text{L} \; e \; \rho \\
\text{loadc} \; (p \ast a) \\
\text{add}
\]

\[
\text{code}_\text{R} \; q \; \rho = \text{loadc} \; q \\
q \text{ constant}
\]

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

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

\[
\text{code}_\text{L}((e_1 \bowtie e_2) \; \rho = \text{code}_\text{R} \; e_1 \; \rho \\
\text{code}_\text{R} \; e_2 \; \rho \\
\text{op} \\
\text{op instruction for operator } \bowtie
\]
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):

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th>free</th>
</tr>
</thead>
</table>

Potential Solutions
- Trust the programmer. Manage freed storage in a particular data structure (free list) \( \rightarrow \) malloc or free may become expensive.
- Do nothing, i.e.:

\[
\text{code free}(e) \cdot \rho = \text{code e \rho pop}
\]

\( \rightarrow \) 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:

\[
\text{code } f \cdot \rho = \text{load}(f) = \text{start address of the code for } f
\]

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

Example

\[
\text{int fac(int x) }
\]

\[
\begin{align*}
\text{if } (x \leq 0) \text{ return } 1; \\
\text{else return } x \cdot \text{fac}(x-1); \\
\end{align*}
\]

\[
\text{main()} \\
\text{n = fac(2) + fac(1);} \\
\text{printf("%d", n);} \\
\]

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.
- 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.
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.
- 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 $\rightarrow$ local variables

FP $\rightarrow$ organizational cells

FPold $\rightarrow$ formal parameters / return value

FP $\leftarrow$ 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 $\langle \text{tag}, a \rangle \in \{G, I\} \times Z$.

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. Before the Function Definitions:
   \[ \rho_0 : \quad i \mapsto (G, 1) \quad l \mapsto (G, 2) \]
   \[ \vdots \]

1. Inside of ith:
   \[ \rho_1 : \quad i \mapsto (L, -4) \quad x \mapsto (L, -3) \quad l \mapsto (G, 2) \quad \text{ith} \mapsto (G, _{ith}) \]
   \[ \vdots \]

Caveat

- The actual parameters are evaluated from right to left!
- The first parameter resides directly below the organizational cells.
- For a prototype \( \tau f (\tau_1, \ldots, \tau_k x_k) \) we define:
  \[ x_1 \mapsto (L, -2 - |\tau_1|) \quad x_k \mapsto (L, -2 - |\tau_1| - \ldots - |\tau_k|) \]

Address Environments Occurring in the Program

0. Before the Function Definitions:
   \[ \rho_0 : \quad i \mapsto (G, 1) \quad l \mapsto (G, 2) \]
   \[ \vdots \]

1. Inside of ith:
   \[ \rho_1 : \quad i \mapsto (L, -4) \quad x \mapsto (L, -3) \quad l \mapsto (G, 2) \quad \text{ith} \mapsto (G, _{ith}) \]
   \[ \vdots \]
Example

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

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

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

Caveat

- The actual parameters are evaluated from right to left II
- The first parameter resides directly below the organizational cells.
- For a prototype \( \tau f(x_1, \ldots, x_n) \) we define:
  \( x_1 \mapsto (L, -2 - |\tau_1|) \quad x_i \mapsto (L, -2 - |\tau_1| - \ldots - |\tau_i|) \)

2. Inside of main:

\[ \begin{align*}
\rho_2 : & \quad i \mapsto (G, 1) \\
       & \quad l \mapsto (G, 2) \\
       & \quad k \mapsto (L, 1) \\
       & \quad ith \mapsto (G, ith) \\
       & \quad main \mapsto (G, main) \\
       & \quad \ldots
\end{align*} \]

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
2. Saving FP, EP
3. Determining the start address of g
4. Setting of the new FP
5. Saving PC and
   Jump to the beginning of g
7. Allocating of local variables

[Note: The actions in curly braces (mark, call, enter, alloc) are part of f]