For every class C we assume that we are given an adress environment ρ_C .

ρ_C maps every identifier x visible inside C to its decorated relative address a . We distingish:

global variable	(G, a)
local variable	(L, a)
attribute	(A, a)
virtual function	(V, b)
non-virtual function	(N, a)
static function	(S, a)

For virtual functions x , we do not store the starting address of the code — but the relative address b of the field of x inside the object :-)

For the various of variables, we obtain for the L-values:

$$\text{code}_{\text{L}} \; x \; \rho \; = \; \begin{cases} \text{loadr } -3 & \text{if} \quad x = \textbf{this} \\[2ex] \text{loadc } a & \text{if} \quad \rho\, x = (G, a) \\[2ex] \text{loadr } a & \text{if} \quad \rho\, x = (L, a) \\[2ex] \text{loadr } -3 & \\ \text{loadc } a & \\ \text{add} & \text{if} \quad \rho\, x = (A, a) \end{cases}$$

In particular, the pointer to the current object has relative address -3 :-)

Accordingly, we introduce the abbreviated operations:

$$
\begin{array}{lcl}
\text{loadm q} & = & \text{loadr } -3 \\
 & & \text{loadc q} \\
 & & \text{add} \\
 & & \text{load} \\
 & & \\
\text{storem q} & = & \text{loadr } -3 \\
 & & \text{loadc q} \\
 & & \text{add} \\
 & & \text{store}
\end{array}
$$

Discussion:

- Besides storing the current object pointer inside the stack frame, we could have additionally used a specific register COP :-)
- This register must updated before calls to non-static member functions and restored after the call.
- We have refrained from doing so since
  - $\rightarrow$ Only some functions are member functions :-)
  - $\rightarrow$ We want to reuse as much of the C-machine as possible :-))

40 Calling Member Functions

Static member functions are considered as ordinary functions :-)

For non-static member functions, we distinguish two forms of calls:

(1) directly: $f\ (e_2, \ldots, e_n)$

(2) relative to an object: $e_1.f\ (e_2, \ldots, e_n)$

Idea:

- The case (1) is considered as an abbreviation of **this**.$f\ (e_2, \ldots, e_n)$:-)
- The object is passed to f as an implicit first argument :-)
- If f is non-virtual, proceed as with an ordinary call of a function :-)
- If f is virtual, insert an indirect call :-)

A non-virtual function:

$$
\begin{array}{lll}
\text{code}_R \; e_1.f\;(e_2,\ldots,e_n)\;\rho & = & \text{code}_R\; e_n\;\rho \\
& & \ldots \\
& & \text{code}_R\; e_2\;\rho \\
& & \text{code}_L\; e_1\;\rho \\
& & \text{mark} \\
& & \text{loadc}\;\_f \\
& & \text{call} \\
& & \text{slide m}
\end{array}
$$

$$
\begin{array}{ll}
\text{where} & (F,\_f) = \rho_C(f) \\
& C = \text{class of } \; e_1 \\
& m = \text{space for the actual parameters}
\end{array}
$$

Note:

The pointer to the object is obtained by computing the L-value of e_1 :-)

A virtual function:

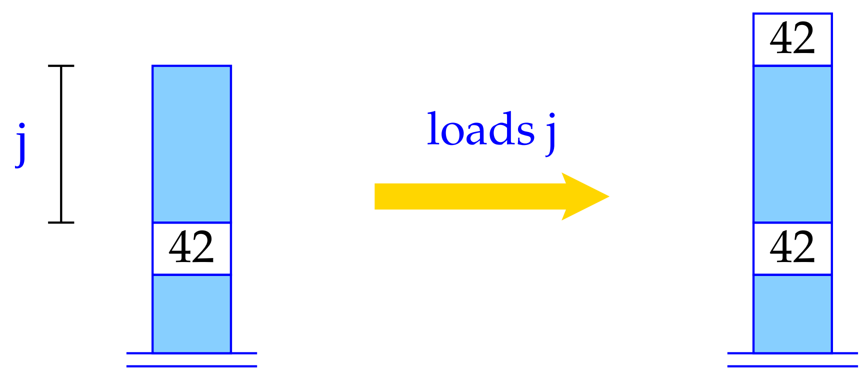
$$
\begin{array}{lll}
\mathrm{code}_\mathrm{R}\ \ e_1.f\ (e_2, \ldots, e_n)\ \ \rho & = & \mathrm{code}_\mathrm{R}\ e_n\ \rho \\
& & \ldots \\
& & \mathrm{code}_\mathrm{R}\ e_2\ \rho \\
& & \mathrm{code}_\mathrm{L}\ \ e_1\ \ \rho \\
& & \mathtt{mark} \\
& & \mathtt{loads}\ 2 \\
& & \mathtt{loadc}\ b \\
& & \mathtt{add}\ ;\ \ \mathtt{load} \\
& & \mathtt{call} \\
& & \mathtt{slide}\ \mathrm{m}
\end{array}
$$

where $\quad (V, b) \;=\; \rho_C(f)$

C = class of $\quad e_1$

m = space for the actual parameters

The instruction loads j loads relative to the stack pointer:



$$S[SP+1] = S[SP-j];$$
$$SP{+}{+};$$

... in the Example:

The recursive call

$$\mathsf{next} \to \mathsf{last}\ ()$$

in the body of the virtual method last is translated into:

loadm 1
mark
loads 2
loadc 2
add
load
call

41 Defining Member Functions

In general, a definition of a member function for class C looks as follows:

$$d \equiv t \;\; f \; (t_2 \; x_2, \ldots, t_n \; x_n) \; \{ \; ss \; \}$$

Idea:

- f is treated like an ordinary function with one extra implicit argument
- Inside f a pointer **this** to the current object has relative address -3 :-)
- Object-local data must be addressed relative to **this** ...

$$
\begin{array}{lllll}
\text{code}_D\ d\ \rho & = & \_f: & \text{enter q} & // \quad \text{Setting the EP} \\
& & & \text{alloc m} & // \quad \text{Allocating the local variables} \\
& & & \text{code}\ ss\ \rho_1 & \\
& & & \text{return} & // \quad \text{Leaving the function}
\end{array}
$$

$$
\begin{array}{llll}
\text{where} & \text{q} & = & maxS + \text{m} \qquad \text{where} \\
& maxS & = & \text{maximal depth of the local stack} \\
& \text{m} & = & \text{space for the local variables} \\
& \text{k} & = & \text{space for the formal parameters (including } \textbf{this}\text{)} \\
& \rho_1 & = & \text{local address environment}
\end{array}
$$

... in the Example:

```
_last:   enter 6          loadm 0      loads 2
         alloc 0          storer -3    loadc 2
         loadm 1          return       add
         loadc 0                       load
         eq          A:   loadm 1      call
         jumpz A          mark         storer -3
                                       return
```

42 Calling Constructors

Every new object should be initialized by (perhaps implicitly) calling a constructor. We distinguish two forms of object creations:

(1) directly: $x = C\,(e_2, \ldots, e_n);$

(2) indirectly: $\textbf{new}\ C\,(e_2, \ldots, e_n)$

Idea for (2):

- Allocate space for the object and return a pointer to it on the stack;
- Initialize the fields for virtual functions;
- Pass the object pointer as first parameter to a call to the constructor;
- Proceed as with an ordinary call of a (non-virtual) member function :-)
- Unboxed objects are considered later ...

$$
\begin{array}{lll}
\text{code}_\text{R}\ \textbf{new}\ C\ (e_2, \ldots, e_n)\ \rho & = & \text{loadc}\ |C| \\
& & \text{new} \\
& & \text{initVirtual}\ \ C \\
& & \text{code}_\text{R}\ e_n\ \rho \\
& & \ldots \\
& & \text{code}_\text{R}\ e_2\ \rho \\
& & \text{loads m} \qquad // \quad \text{loads relative to SP} \quad \text{:-)} \\
& & \text{mark} \\
& & \text{loadc \_C} \\
& & \text{call} \\
& & \text{pop m} + 1
\end{array}
$$

where m = space for the actual parameters.

Before calling the constructor, we initialize all fields of virtual functions.

The pointer to the object is copied into the frame by an extra instruction :-)

Assume that the class C lists the virtual functions $f_1, \ldots, f_r$ for C with the offsets and initial addresses: b_i and a_i, respectively:

Then:

$$
\begin{array}{lll}
\text{initVirtual } C & = & \text{dup} \\
 & & \text{loadc } b_1 \text{ ; add} \\
 & & \text{loadc } a_1 \text{ ; store} \\
 & & \text{pop} \\
 & & \ldots \\
 & & \text{dup} \\
 & & \text{loadc } b_r \text{ ; add} \\
 & & \text{loadc } a_r \text{ ; store} \\
 & & \text{pop}
\end{array}
$$

43 Defining Constructors

In general, a definition of a constructor for class C looks as follows:

$$d \equiv C\ (t_2\ x_2, \ldots, t_n\ x_n)\ \{\ ss\ \}$$

Idea:

- Treat the constructor as a definition of an ordinary member function :-)

... in the Example:

```
_list:  enter 3     loada 1     loadc 0
        alloc 0     loadc 1     storem 1
        loadr -4    add         pop
        storem 0    storea 1    return
        pop         pop
```

Discussion:

The constructor may issue further constructors for attributes if desired :-)

The constructor may call a constructor of the super class B as first action:

$$
\begin{array}{lll}
\text{code } B\,(e_2, \ldots, e_n);\ \rho & = & \text{code}_R\ e_n\ \rho \\
& & \quad\ldots \\
& & \text{code}_R\ e_2\ \rho \\
& & \text{loadr } -3 \\
& & \text{mark} \\
& & \text{loadc } \_B \\
& & \text{call} \\
& & \text{pop m} + 1
\end{array}
$$

where m = space for the actual parameters.

The constructor is applied to the current object of the calling constructor!

44 Initializing Unboxed Objects

Problem:

The same constructor application can be used for initializing several variables:

$$x = x_1 = C\ (e_2, \ldots, e_n)$$

Idea:

- Allocate sufficient space for a temporary copy of a new C object.
- Initialize the temporary copy.
- Assign this value to the variables to be intialized :-)

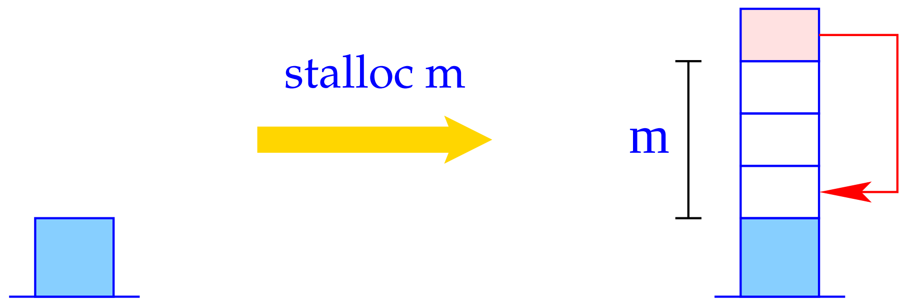
$$
\begin{array}{lll}
\mathrm{code_R}\ \ C\ (e_2, \ldots, e_n)\ \rho & = & \mathrm{stalloc}\ |C| \\
 & & \mathrm{initVirtual}\ C \\
 & & \mathrm{code_R}\ e_n\ \rho \\
 & & \ldots \\
 & & \mathrm{code_R}\ e_2\ \rho \\
 & & \mathrm{loads\ m} \\
 & & \mathrm{mark} \\
 & & \mathrm{loadc\ \_C} \\
 & & \mathrm{call} \\
 & & \mathrm{pop\ m} + 2
\end{array}
$$

where m = space for the actual parameters.

Note:

The instruction stalloc m is like malloc m but allocates on the stack :-)

We assume that we have assignments between complex types :-)



$$SP = SP+m+1;$$

$$S[SP] = SP-m;$$