41 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: \( f(e_2, \ldots, e_n) \)

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{align*}
\text{code}_e & \quad e_1, f(e_2, \ldots, e_n) \quad \rho \\
\text{code}_e & \quad e_2 \quad \rho \\
\text{mark} & \\
\text{load} & f \\
\text{call} & \\
\text{slide} & m \\
\end{align*}
```

where \( (f, f') = \rho_e(f) \)

- \( C = \text{class of } e_1 \)
- \( m = \text{space for the actual parameters} \)

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

A virtual function:

```
\begin{align*}
\text{code}_e & \quad e_1, f(e_2, \ldots, e_n) \quad \rho \\
\text{code}_e & \quad e_2 \quad \rho \\
\text{mark} & \\
\text{load} & 2 \\
\text{load} & b \\
\text{add} & \text{load} \\
\text{call} & \\
\text{slide} & m \\
\end{align*}
```

where \( (V, b) = \rho_e(f) \)

- \( C = \text{class of } e_1 \)
- \( m = \text{space for the actual parameters} \)
42 Defining Member Functions

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

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

Idea:

- $f$ is treated like an ordinary function with one extra implicit argument
- Inside $f$ a pointer $\text{this}$ to the current object has relative address $\text{this}$
- Object-local data must be addressed relative to $\text{this}$...
... in the Example:

\[
\begin{align*}
\_\text{last} & : \text{enter 6} & \text{loadm 0} & \text{loads 2} \\
& : \text{alloc 0} & \text{storer -3} & \text{loadc 2} \\
& : \text{loadm 1} & \text{return} & \text{add} \\
& : \text{loadc 0} & \text{load} & \text{eq} \\
& : \text{jump A} & \text{mark} & \text{storer -3} \\
& : \text{loadm 1} & \text{call} & \text{return}
\end{align*}
\]

A virtual function:

\[
\begin{align*}
\text{code}_c \left( e_1 f \left( e_2, \ldots, e_n \right) \rho \right) &= \text{code}_c e_n \rho \\
\vdots \\
\text{code}_c e_2 \rho \\
\text{code}_c e_1 \rho \\
\text{mark} \\
\text{loads 2} \\
\text{loadc b} \\
\text{add} : \text{load} \\
\text{call} \\
\text{slide m}
\end{align*}
\]

where \((V, b) = \rho_c(f)\)

\[
\begin{align*}
C &= \text{class of } e_1 \\
m &= \text{space for the actual parameters}
\end{align*}
\]

43 Calling Constructors

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

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

(2) indirectly: \( \text{new } C \left( e_2, \ldots, e_n \right) \)

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 (\( \leftarrow \))
- Unboxed objects are considered later...
code_{new} C (e_2, \ldots, e_n) \rho = \text{loadc } [C] \\
  \text{new} \\
  \text{initVirtual } C \\
  \text{code}_{e_n} \rho \\
  \ldots \\
  \text{code}_{e_2} \rho \\
  \text{loads } m \quad \text{// loads relative to SP } \rightarrow \\
  \text{mark} \\
  \text{loadc } \_C \\
  \text{call} \\
  \text{pop } m + 1 \\
 \text{where } m = \text{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 \rightarrow 

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{align*}
\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{align*}
44 Defining Constructors

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

$$d = C \{ x_1, x_2, \ldots, x_n \} \{ \text{ss} \}$$

Idea:

- Treat the constructor as a definition of an ordinary member function $\Rightarrow$

---

... in the Example:

```
_list: enter 3     load 1  load 0
 alloc 0  load c 1  store 1
 loadr 4  add     pop
 store 0  store 1 return
 pop
```

---

44 Defining Constructors

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

$$d = C \{ x_1, x_2, \ldots, x_n \} \{ \text{ss} \}$$

Idea:

- Treat the constructor as a definition of an ordinary member function $\Rightarrow$

---

40 Object Layout

Idea:

- Only attributes and virtual member functions are stored inside the class $!!$
- The addresses of non-virtual or static members as well as of constructors can be resolved at compile time $!!$
- The fields of a sub-class are appended to the corresponding fields of the super-class ...

... in our Example:

```
info
next
last
```
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:

$$\text{code } B (e_2, \ldots, e_n) \rho = \begin{array}{l}
\text{code}_B e_2 \rho \\
\ldots \\
\text{code}_B e_n \rho \\
\text{loads } m \\
\text{mark} \\
\text{load}_B \\
\text{call} \\
\text{pop } m + 1 \\
\end{array}$$

where $m =$ space for the actual parameters.
The constructor is applied to the current object of the calling constructor!

45 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 initialized :-)

Note:
The instruction `stalloc m` is like `malloc m` but allocates on the stack :-)
We assume that we have assignments between complex types :-)

$\text{SP} = \text{SP} + m + 1$
$\text{S[SP]} = \text{SP} - m$
stalloc m

SP = SP+m+1;
S(SP) = SP–m;