39 Object Layout

Idea:

- Only attributes and virtual member functions are stored inside the class.
- The addresses of non-virtual or static member functions 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
moreInfo
```
For every class \( C \) we assume that we are given an address environment \( \rho_C \). \( \rho_C \) maps every identifier \( x \) visible inside \( C \) to its decorated relative address \( a \). We distinguish:

- **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.

Accordingly, we introduce the abbreviated operations:

\[
\begin{align*}
\text{loadm } q &= \text{loadr} - 3 \\
\text{loadc } q &= \text{loadr} - 3 \\
\text{add} &= \text{loadr} - 3 \\
\text{load} &= \text{loadr} - 3 \\
\text{storem } q &= \text{loadr} - 3 \\
\text{loadc } q &= \text{loadr} - 3 \\
\text{add} &= \text{loadr} - 3 \\
\text{store} &= \text{loadr} - 3
\end{align*}
\]

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

\[
\text{code}_1 \ x \ \rho = \begin{cases} 
\text{loadr} - 3 & \text{if } x = \text{this} \\
\text{loadc } q & \text{if } \rho x = (G, a) \\
\text{loadc } q & \text{if } \rho x = (L, a) \\
\text{loadc } q & \text{if } \rho x = (A, a) \\
\text{add} & \text{if } \rho x = (S, a) \\
\end{cases}
\]

In particular, the pointer to the current object has relative address \(-3\) (\(>)\).
Accordingly, we introduce the abbreviated operations:

\[
\begin{align*}
\text{loadm } q &= \text{loadr } - 3 \\
\text{loadc } q &= \\
\text{add} &= \\
\text{load} &= \\
\text{storem } q &= \text{loadr } - 3 \\
\text{loadc } q &= \\
\text{add} &= \\
\text{store} &= \\
\end{align*}
\]

Discussion:

- Besides storing the current object pointer inside the stack frame, we could have additionally used a specific register \texttt{COP}.
- This register must be updated before calls to non-static member functions and restored after the call.
- We have refrained from doing so since
  - Only some functions are member functions.
  - 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 \( \text{this}.f(e_2, \ldots, e_n) \)
- 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:

\[
\text{code}_\kappa e_1.f(e_2, \ldots, e_n) \rho = \text{code}_\kappa e_n \rho \\
\vdots \\
\text{code}_\kappa e_2 \rho \\
\text{code}_\kappa e_1 \rho \\
\text{mark} \\
\text{loads 2} \\
\text{loadc b} \\
\text{add} ; \text{load} \\
\text{call} \\
\text{slide m} \\
\]  

where \( (V,b) = \rho \circ (f) \)
- \( C = \text{class of } e_1 \)
- \( m = \text{space for the actual parameters} \)

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

A virtual function:

\[
\text{code}_\kappa e_1.f(e_2, \ldots, e_n) \rho = \text{code}_\kappa e_n \rho \\
\vdots \\
\text{code}_\kappa e_2 \rho \\
\text{code}_\kappa e_1 \rho \\
\text{mark} \\
\text{loads 2} \\
\text{loadc f} \\
\text{call} \\
\text{slide m} \\
\]  

where \( (V,b) = \rho \circ (f) \)
- \( C = \text{class of } e_1 \)
- \( m = \text{space for the actual parameters} \)

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

\[
\text{code}_{\kappa_1.f}(e_2, \ldots, e_n, \rho) = \text{code}_{\kappa_n, \rho} \\
\vdots \\
\text{code}_{\kappa_2, \rho} \\
\text{code}_{\kappa_1, \rho} \\
\text{mark} \\
\text{loads 2} \\
\text{loadc} \, b \\
\text{add } \; \text{load} \\
\text{call} \\
\text{slide} \, m \\
\text{where} \quad (V, b) = \nu_{\kappa}(f) \\
(C = \text{class of } e_1) \\
m = \text{space for the actual parameters}
\]

... in the Example:

The instruction \text{loads } j \text{ loads relative to the stack pointer: }

\[
\text{42} \\
\text{42}
\]

in the body of the virtual method \text{last} is translated into:

\[
\text{ldxm 1} \\
\text{mark} \\
\text{loads 2} \\
\text{loadc 2} \\
\text{add} \\
\text{load} \\
\text{call}
\]
... in the Example:

The recursive call

\[ \text{next} \rightarrow \text{last}() \]

in the body of the virtual method \( \text{last} \) is translated into:

\[
\begin{align*}
\text{loadm} & \ 1 \\
\text{mark} & \\
\text{loads} & \ 2 \\
\text{loade} & \ 2 \\
\text{add} & \\
\text{load} & \\
\text{call} & \\
\end{align*}
\]

\[ \text{>>} \]

### 41 Defining Member Functions

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

\[
d = t \ f \ (t_2, s_2, \ldots, l_n, x_n) \ss
\]

**Idea:**

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

\[ \text{370} \]

\[
\begin{align*}
\text{codeD} \ d \ \rho & = \ _f:\ \text{enter} \ q & \quad & \text{Setting the EP} \\
& \quad \text{alloc} \ m & \quad & \text{Allocating the local variables} \\
& \quad \text{code} \ s s \ \rho_1 & \quad & \text{Leaving the function}
\end{align*}
\]

**where**

- \( q = maxS + m \)
- \( maxS = \) maximal depth of the local stack
- \( m = \) space for the local variables
- \( k = \) space for the formal parameters (including \texttt{this})
- \( \rho_1 = \) local address environment

\[ \text{371} \]
... in the Example:

\[ \_\text{last: enter 6} \]
\[ \text{ alloc 0} \]
\[ \text{ loadm 1} \]
\[ \text{ loadc 0} \]
\[ \text{ eq} \]
\[ \text{ jump A} \]
\[ \text{ loadm 0} \]
\[ \text{ loadc 2} \]
\[ \text{ return} \]
\[ \text{ add} \]
\[ \text{ load} \]
\[ \text{ call} \]
\[ \text{ storer -3} \]
\[ \text{ return} \]

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: \( \text{new } C (e_2, \ldots, e_n) \)

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 \( \Rightarrow \)
- Unboxed objects are considered later ...

Assume that the class \( C \) lists the virtual functions \( f_1, \ldots, f_r \) for \( C \) with the offsets and initial addresses \( a_1 \) and \( a_2, \ldots, a_r \), respectively:

Then:

\[
\text{initVirtual } C = \begin{array}{c}
dup \\
\text{loadc } b_1; \text{ add} \\
\text{loadc } f; \text{ store} \\
\text{pop} \\
\text{...} \\
\text{dup} \\
\text{loadc } b_2; \text{ add} \\
\text{loadc } f; \text{ store} \\
\text{pop} \\
\end{array}
\]

Note:

Before calling the constructor, we initialize all fields of virtual functions.
The pointer to the object is copied into the frame by a new instruction \( \Rightarrow \)
43 Defining Constructors

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

$$d \equiv C(t_1, x_2, \ldots, t_n, x_n) \{ \ldots \}$$

Idea:

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

Discussion:

The constructor may issue further constructors for attributes if desired $\Rightarrow$
The constructor may call a constructor of the super class $B$ as first action:

$$\text{code } B(t_1, \ldots, t_n); \rho = \text{code}_B(t_n \rho)
\ldots
\text{code}_B(t_2 \rho)
\text{loadr } -3
\text{mark}
\text{loadc } _B
\text{call}
\text{pop } m + 1$$

... in the Example:

<table>
<thead>
<tr>
<th>_list</th>
<th>enter 3</th>
<th>loada 1</th>
<th>loadc 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>alloc 0</td>
<td>loadc 1</td>
<td>storem 1</td>
<td></td>
</tr>
<tr>
<td>loadr -4</td>
<td>add</td>
<td>pop</td>
<td></td>
</tr>
<tr>
<td>storem 0</td>
<td>storea 1</td>
<td>return</td>
<td></td>
</tr>
<tr>
<td>pop</td>
<td>pop</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

where $m =$ space for the actual parameters.

Thus, the constructor is applied to the current object of the calling constructor $\Rightarrow$
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 initialized.

Note:
The instruction \texttt{stloc m} is like \texttt{malloc m} but allocates on the stack. We assume that we have assignments between complex types. :)

\[ \text{code}_R(C(e_2, \ldots, e_n) \rho) = \text{stloc } |C| \]
\[ \text{initVirtual C} \]
\[ \text{code}_R e_1 \rho \]
\[ \ldots \]
\[ \text{code}_R e_2 \rho \]
\[ \text{loads } m + 1 \]
\[ \text{mark} \]
\[ \text{loade } \_C \]
\[ \text{call} \]
\[ \text{pop } m + 1 \]

where \( m \) = space for the actual parameters.

Note:
The instruction \texttt{stloc m} is like \texttt{malloc m} but allocates on the stack. We assume that we have assignments between complex types. :)

\[ \text{code}_R(C(e_2, \ldots, e_n) \rho) = \text{stloc } |C| \]
\[ \text{initVirtual C} \]
\[ \text{code}_R e_1 \rho \]
\[ \ldots \]
\[ \text{code}_R e_2 \rho \]
\[ \text{loads } m + 1 \]
\[ \text{mark} \]
\[ \text{loade } \_C \]
\[ \text{call} \]
\[ \text{pop } m + 2 \]

where \( m \) = space for the actual parameters.

Note:
The instruction \texttt{stloc m} is like \texttt{malloc m} but allocates on the stack. We assume that we have assignments between complex types. :)

\[ SP = SP+m+1; \]
\[ S[SP] = SP-m; \]
\[
\text{code}_\varrho C (e_2, \ldots, e_n) \varrho = \text{stalloc} |C| \\
\text{initVirtual} C \\
\text{code}_\varrho e_1 \varrho \\
\ldots \\
\text{code}_\varrho e_2 \varrho \\
\text{loads} m + 1 \\
\text{mark} \\
\text{loadc}_C \\
\text{call} \\
\text{pop} m + 1
\]

where \( m \) = space for the actual parameters.

**Note:**
The instruction \( \text{stalloc} \varrho \) is like \( \text{malloc} \varrho \) but allocates on the stack. 

We assume that we have assignments between complex types.

---

**Threads**

\[ C : I_1, I_2 \]
\( \forall \text{ Integers} \)

\[ C : I_1, I_2 \]

Threads

Threads