Discussion:

- We adopt the C++ perspective on classes and objects.
- We extend our implementation of C. In particular...
- Classes are considered as extensions of structs. They may comprise:
  - attributes, i.e., data fields;
  - constructors;
  - member functions which either are virtual, i.e., are called depending on the run-time type or non-virtual, i.e., called according to the static type of an object;
  - static member functions which are like ordinary functions
- We ignore visibility restrictions such as public, protected or private but simply assume general visibility.
- We ignore multiple inheritance.

Example:

```c
int count = 0;
class list {
  int info;
  list * next;
  list (int x) {
    info = x; count++; next = null;
  }

  virtual int last () {
    if (next == null) return info;
    else return next -> last();
  }
}
```
40 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:

![Diagram showing fields info, next, last]

Example (cont.):
- The fields of a sub-class are appended to the corresponding fields of the super-class.

```cpp
class mylist : list {
    int moreInfo;
}
```

... results in:

![Diagram showing fields 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.

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.
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:

<table>
<thead>
<tr>
<th>Type</th>
<th>(G, a)</th>
<th>(L, a)</th>
<th>(A, a)</th>
<th>(V, b)</th>
<th>(N, a)</th>
<th>(S, a)</th>
</tr>
</thead>
<tbody>
<tr>
<td>global variable</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>local variable</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>attribute</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>virtual function</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>non-virtual function</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>static function</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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. $\sim$

Accordingly, we introduce the abbreviated operations:

$$
\begin{align*}
\text{loadm } q &= \text{loadr } -3 \\
                & \text{loadc } q \\
                & \text{add} \\
\text{storem } q &= \text{loadr } -3 \\
                & \text{loadc } q \\
                & \text{add} \\
\end{align*}
$$

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

$$
\begin{align*}
\text{code } x \rho &= \begin{cases} 
\text{loadr } -3 & \text{if } x = \text{this} \\
\text{loadc } a & \text{if } \rho x = (G, a) \\
\text{loadr } -3 & \text{if } \rho x = (L, a) \\
\text{loadc } a & \text{if } \rho x = (A, a) \\
\text{add} & \text{if } \rho x = (S, a)
\end{cases}
\end{align*}
$$

In particular, the pointer to the current object has relative address $-3$. $\sim$

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

$$
\begin{align*}
\text{code } x \rho &= \begin{cases} 
\text{loadr } -3 & \text{if } x = \text{this} \\
\text{loadc } a & \text{if } \rho x = (G, a) \\
\text{loadr } -3 & \text{if } \rho x = (L, a) \\
\text{loadc } a & \text{if } \rho x = (A, a) \\
\text{add} & \text{if } \rho x = (S, a)
\end{cases}
\end{align*}
$$

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

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

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

\[
\text{code }_x \rho = \begin{cases} 
\text{loadr } -3 & \text{if } x = \text{this} \\
\text{load } a & \text{if } \rho x = (C, a) \\
\text{load } a & \text{if } \rho x = (L, a) \\
\text{loadr } -3 \\
\text{load } a \\
\text{add} & \text{if } \rho x = (A, a)
\end{cases}
\]

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

Discussion:

- Besides storing the current object pointer inside the stack frame, we could have additionally used a specific register \text{COP} \to
- This register must 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 \to
  → We want to reuse as much of the C-machine as possible \to
41 Calling Member Functions

Static member functions are considered as ordinary functions \( \Rightarrow \).

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 \( \text{this}.f(e_2, \ldots, e_n) \) \( \Rightarrow \).
- The object is passed to \( f \) as an implicit first argument \( \Rightarrow \).
- If \( f \) is non-virtual, proceed as with an ordinary call of a function \( \Rightarrow \).
- If \( f \) is virtual, insert an indirect call \( \Rightarrow \).

Discussion:

- Besides storing the current object pointer inside the stack frame, we could have additionally used a specific register \( \text{COP} \) \( \Rightarrow \).
- 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 \( \Rightarrow \).
  - We want to reuse as much of the C-machine as possible \( \Rightarrow \).

A non-virtual function:

\[
\begin{align*}
\text{code}_{\pi} e_1.f(e_2, \ldots, e_n) \ p & = \text{code}_{\pi} e_n \ p \\
& \quad \text{...} \\
& \quad \text{code}_{\pi} e_2 \ p \\
& \quad \text{code}_{\pi} e_1 \ p \\
& \quad \text{mark} \\
& \quad \text{loadc} \ f \\
& \quad \text{call} \\
& \quad \text{slide} \ m
\end{align*}
\]

where \( (F, -f) = \rho_C(f) \) \( \Rightarrow \).

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

Note:

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

$$\text{code}_\rho e_1.f(e_2, \ldots, e_n) = \text{code}_\rho e_a \; \rho$$

...  
\text{code}_\rho e_2 \; \rho  
\text{code}_\rho e_1 \; \rho  
\text{mark}  
\text{loadc} \_f  
\text{call}  
\text{slide} \; m  

where \( (f).f = \rho_c(f) \)  
\( C = \text{class of} \; e_1 \)  
\( m = \text{space for the actual parameters} \)

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

(A page number 380)

A virtual function:

$$\text{code}_\rho e_1.f(e_2, \ldots, e_n) = \text{code}_\rho e_v \; \rho$$

...  
\text{code}_\rho e_2 \; \rho  
\text{code}_\rho e_1 \; \rho  
\text{mark}  
\text{loads} 2  
\text{loadc} \; b  
\text{add} ; \text{load}  
\text{call}  
\text{slide} \; m  

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

(A page number 381)

A non-virtual function:

$$\text{code}_\rho e_1.f(e_2, \ldots, e_n) = \text{code}_\rho e_a \; \rho$$

...  
\text{code}_\rho e_2 \; \rho  
\text{code}_\rho e_1 \; \rho  
\text{mark}  
\text{loadc} \_f  
\text{call}  
\text{slide} \; m  

where \( (f).f = \rho_c(f) \)  
\( C = \text{class of} \; e_1 \)  
\( m = \text{space for the actual parameters} \)

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

(A page number 380)

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

\[ \text{SP}+1 = \text{SP}+j; \]  
\( \text{SP}++ \); 

(A page number 382)
42 Defining Member Functions

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

\[ d \equiv \text{abstract} \{ x_1, x_2, \ldots, x_n \} \{ \text{ss} \} \]

Idea:

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

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

The recursive call

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

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

\[ \text{loadm 1} \]
\[ \text{mark} \]
\[ \text{loades 2} \]
\[ \text{loadc 2} \]
\[ \text{add} \]
\[ \text{load} \]
\[ \text{call} \]

\[ \text{code}_{22} \quad d \rho = \text{...} : \text{enter q} \quad / / \text{Setting the EP} \]
\[ \text{alloc m} \quad / / \text{Allocating the local variables} \]
\[ \text{code ss \rho_l} \]
\[ \text{return} \quad / / \text{Leaving the function} \]

where \( q \) = \( \text{maxS} + \text{m} \)

where

\( \text{maxS} = \) maximal depth of the local stack
\( \text{m} = \) space for the local variables
\( k = \) space for the formal parameters (including \textit{this})
\( \rho_l = \) local address environment
... in the Example:

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

43 Calling Constructors

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

(1) directly: \( x = C(e_2, \ldots, e_n) \);
(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.
- Unboxed objects are considered later...

```plaintext
codeC new C(e_2, \ldots, e_n) \rho =
   loade [C]
   new
   initVirtual C
codeC e_n \rho
   ...
codeC e_2 \rho
   loade m // loads relative to SP
   mark
   loade _C
   call
   pop m + 1
```

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...
44 Defining Constructors

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

\[
d \in C \left( x_2, \ldots, x_n \right) \{ \cdots \}
\]

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:

\[
\text{code } B \left( \epsilon_2, \ldots, \epsilon_n \right); \rho = \text{code}_{\rho} \epsilon_n \rho \\
\text{...}
\]

\[
\text{code}_{\rho} \epsilon_2 \rho \\
\text{loadr } -3 \\
\text{mark} \\
\text{loadc } _B \\
\text{call} \\
\text{pop } m + 1
\]

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:

\[
x = x_1 = C \left( \epsilon_2, \ldots, \epsilon_n \right)
\]

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 \( \Rightarrow \)
\[
\text{code}_{C} (e_2, \ldots, e_n) \rho = \text{stalloca}_{C}
\]
\[
\text{initVirtual}_{C}
\]
\[
\text{code}_{e_n} \rho
\]
\[
\ldots
\]
\[
\text{code}_{e_2} \rho
\]
\[
\text{loads}_{m}
\]
\[
\text{mark}
\]
\[
\text{load}_{C}
\]
\[
\text{call}
\]
\[
\text{pop}_{m + 2}
\]

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

**Note:**

The instruction \text{stalloca}_{m} \ is like \text{malloc}_{m} \ but allocates on the stack \( \Rightarrow \)

We assume that we have assignments between complex types \( \Rightarrow \)
46 The Language ThreadedC

We extend C by a simple thread concept. In particular, we provide functions for:

- generating new threads: `create();`
- terminating a thread: `exit();`
- waiting for termination of a thread: `join();`
- mutual exclusion: `lock(), unlock();` ...

In order to enable a parallel program execution, we extend the virtual machine (what else? :-)}