

Classes and Objects

Example:

```
int count = 0;
class list {
    int info;
    class list * next;
    list (int x) {
        info = x; count++; next = null;
    }
    virtual int last () {
        if (next == null) return info;
        else return next → last ();
    }
}
```

Discussion:

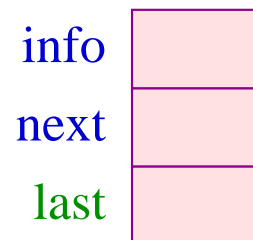
- We adopt the C++ perspective on classes and objects.
- We extend our implementation of C. In particular ...
- Classes are considered as extensions of **structs**. They may comprise:
 - ⇒ attributes, i.e., data fields;
 - ⇒ constructors;
 - ⇒ member functions which either are **virtual**, i.e., are 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 :-)

50 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:



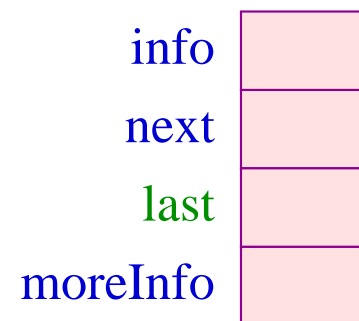
Idea (cont.):

- The fields of a sub-class are **appended** to the corresponding fields of the super-class :-)

Example:

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

... results in:



For every class C we assume that we are given an **address environment** ρ_C .
 ρ_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 the various of variables, we obtain for the **L-values**:

$$\text{code}_L x \rho = \left\{ \begin{array}{ll} \text{loadr } 1 & \text{if } x = \mathbf{this} \\ \text{loadc } a & \text{if } \rho x = (G, a) \\ \text{loadr } a & \text{if } \rho x = (L, a) \\ \text{loadr } 1 \\ \text{loadc } a \\ \text{add} & \text{if } \rho x = (A, a) \end{array} \right.$$

In particular, the pointer to the current object has relative address 1 :-)

Accordingly, we introduce the abbreviated operations:

loadm q = loadr 1
 loadc q
 add
 load

bla ; storem q = loadr 1
 loadc q
 add
 bla
 store

Discussion:

- Besides storing the current object pointer inside the stack frame, we could have additionally used a specific **register** *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 :-))

51 Calling Member Functions

Static member functions are considered as ordinary functions :-)

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

- (1) directly: $f(e_2, \dots, e_n)$
- (2) relative to an object: $e_1.f(e_2, \dots, e_n)$

Idea:

- The case (1) is considered as an abbreviation of $\mathbf{this}.f(e_2, \dots, e_n)$:-)
- 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:

```
codeR e1.f (e2, ..., en) ρ = mark
                                codeL e1 ρ
                                codeR e2 ρ
                                ...
                                codeR en ρ
                                loadc f
                                call m + 1
```

where $(F, _f) = \rho_C(f)$

C = class of e_1

m = space for the actual parameters

Note:

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

A virtual function:

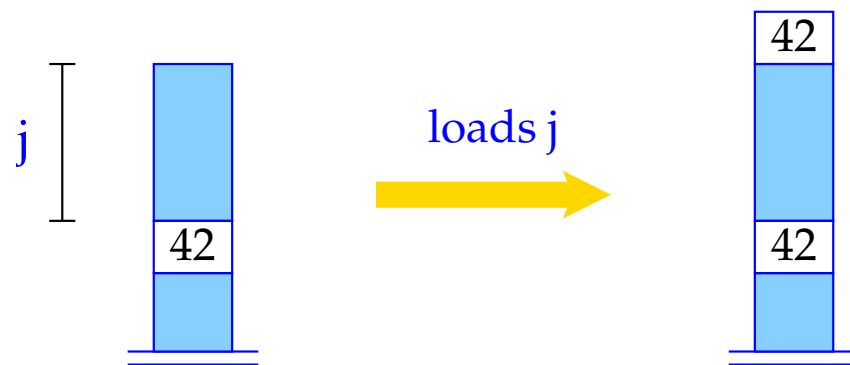
$\text{code}_R e_1.f(e_2, \dots, e_n) \rho =$ mark
code_L $e_1 \rho$
code_R $e_2 \rho$
...
code_R $e_n \rho$
loads m
loadc b
add ; load
call $m + 1$

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

$C =$ class of 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

`next` → `last` ()

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

`mark`

`loadm 1`

`loads 0`

`loadc 2`

`add`

`load`

`call 1`

52 Defining Member Functions

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

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

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 1 :-)
- Object-local data must be addressed relative to **this** ...

```

codeD d ρ = f : enter q      // Setting the EP
                alloc m        // Allocating the local variables
                codeSS ρ1
                return          // Leaving the function

```

where q = $maxS + m$ where
 $maxS$ = maximal depth of the local stack
 m = space for the local variables
 k = space for the formal parameters (including **this**)
 ρ_1 = local address environment

... in the Example:

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

53 Calling Constructors

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

- (1) directly: $x = C(e_2, \dots, e_n);$
- (2) indirectly: **new** $C(e_2, \dots, e_n)$

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 ...

```

codeR new C (e2, ..., en); ρ = malloc |C|
                               initVirtual C
                               mark
                               loads 4 // loads relative to SP :-)
                               codeR e2 ρ
                               ...
                               codeR en ρ
                               loadc _C
                               call m + 1
                               pop

```

where m = space for the actual parameters.

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 :-)

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

Then:

```
initVirtual C = dup
                loadc  $b_1$  ; add
                loadc  $a_1$  ; store
                pop
                ...
                dup
                loadc  $b_r$  ; add
                loadc  $a_r$  ; store
                pop
```

54 Defining Constructors

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

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

Idea:

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

... in the Example:

```
_list:  enter 3      loada 1      loadc 0
        alloc 0   dup          storem 1
        loadr 2   loadc 1      pop
        storem 0  add          return
        pop      storea 1
                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:

```
code  $B(e_2, \dots, e_n); \rho =$  mark  
                                loadr 1  
                                codeR  $e_2 \rho$   
                                ...  
                                codeR  $e_n \rho$   
                                loadc  $_B$   
                                call m
```

where $m =$ space for the actual parameters.

Thus, the constructor is applied to the current object of the calling constructor :-)

55 Initializing Unboxed Objects

Problem:

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

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

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 :-)

```

codeR C (e2, ..., en) ρ = stalloc |C|
                             initVirtual C
                             mark
                             loads 4
                             codeR e2 ρ
                             ...
                             codeR en ρ
                             loadc _C
                             call m + 1
                             pop
                             pop

```

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 :-)



$SP = SP + m + 1;$

$S[SP] = SP - m;$