

**The Structure of Typed  
Programming Languages**  
 (Chapter 4 cont.)

CS386L- Programming Languages  
 Dr. Greg Lavender  
 Department of Computer Sciences  
 The University of Texas at Austin

CS386L: Lecture-12 - The Structure of Typed Programming Languages: Chapter 4 cont.

## Basic Classes and Objects

- Type-structure blocks whose bodies are records can be used as a foundation for classes
- Example:
  - class STACK
    - begin // private variables
    - var CTR:newint, var STK: array[1..100] of newint
    - in record // public methods
    - proc PUSH (X:intexp) = if @CTR=100 then skip
    - else CTR := CTR+1;
    - STK[@CTR] := X
    - fi,
    - proc POP = if @CTR=0 then skip else CTR := @CTR - 1 fi,
    - fun TOP = if @CTR = 0 then fail
    - else @(STK[@CTR]) fi
    - proc INIT = CTR := 0 // a constructor
    - end
    - end

4/19/04
2

## Instances of Classes

- When we declare an instance of a class, we expect to get a copy of the local variables, i.e., an "object"
  - recall that a type-structure returns a "record value"
    - since we use the qualification principle to "hide" local variables and we group procs/funs into a record we have an interface, i.e., we have an object
    - var A:STACK
      - allocates storage for local vars CTR and STK
      - call A.INIT; call A.PUSH(0); call A.POP; ...; A.TOP; ...
      - if we declare var B:STACK, we get another (unique) record value as a result, with its own local copies of CTR and STK
      - the l-value of the record value is effectively the "this" reference to the object (ala Java, or this pointer ala C++)

4/19/04

3

## Extending Classes

- Another key feature of OOP, is composition of classes
  - there are two ways to do composition
    - embed an instance of a class in a new object
    - extend a class by inheriting from it
    - example:
 

```
class EXTENDED-STACK =
begin var M:STACK, var N:newint
in record
  proc P = .... // methods unique to extended stack
  // 'wrapper' methods that delegate to embedded stack
  proc PUSH-EXTENDED(X:intexp) = call M.PUSH(X),
  ...
end
end
```

4/19/04

4

## Basic Inheritance

- Inheritance of class structure
  - sometimes we want to extend, not hide, the interface of the inherited class by adding new methods
    - a form of "subclassing" or interface inheritance
      - but without any kind of "protected" scope for locals such that locals in the base class are not visible to procs/funs (methods) in the extended class
      - class EXTENDED-STACK = inherits STACK with
 

```
begin var N: newint, ...
          in record proc P = ...
            end
        end
      end
```

4/19/04

5

## Scoping Issues

- Composition via inheritance brings into focus many scope resolution issues
  - public vs private vs protected visibility
  - name conflicts
    - how do we handle proc/fun name conflicts?
    - if we allow overloaded proc/fun definitions, how do we distinguish which one to call in an invocation?
    - example:
      - class T = record proc P = ... end;
      - class U = inherits T with record proc P = ... end;
      - var X:U in call X.P // which P do we invoke: T.P or U.P?
      - we need a rule or set of rules, for example, since X is of type U invoke U.P since it is the most direct (most local) proc
      - this brings into play type information as a way to decide what to do. This also means we need to think about subtyping.

4/19/04

6

## Simple Typing of Class Inheritance

- typing of single class inheritance

$$\frac{\pi \vdash T_1 : \pi_1 \text{ class} \quad \pi \cup \pi_1 \vdash T_2 : \pi_2 \text{ class}}{\pi \vdash \text{inherits } T_1 \text{ with } T_2 : (\pi_1 \cup \pi_2) \text{ class}}$$

4/19/04

7

## Multiple Inheritance

- The scope & name resolution situation becomes more complicated if we allow multiple inherited base classes
  - C++ has complex rules for multiple inheritance resolution
  - Java avoids multiple class inheritance, but allows multiple interface inheritance
  - Example
    - class T = record proc P = ... end;
    - class U = record proc P = ... end;
    - class R = inherits T, inherits U with ...;
    - var X:R in call X.P
    - which proc does X.P invoke?

4/19/04

8

## Object-Oriented Programming

- What features make a language object-oriented?
  - OOP = classes + objects + inheritance
    - classes - a way to define new types
    - objects - unique instances of those types
    - inheritance - a way to compose classes (types)
      - interface inheritance (subtyping or "is-a" inheritance)
      - implementation inheritance (use inheritance)
    - generics are often useful too
      - template classes in C++
        - especially for implementing container types
        - `stack<int>` vs `stack<string>`
    - other key issues
      - scope rules
      - name overloading
      - static vs dynamic typing
      - static vs dynamic method dispatching
        - i.e., virtual methods in C++

4/19/04

9

## Objects and "Self"

- Object-oriented languages typically define a special keyword to hold the l-value of an object
  - every local variable inside of a class has an l-value that is relative to the "self" variable and methods of a class access an object using the self variable
  - smalltalk originally called this value "self"
  - there is an object-oriented language called Self
    - <http://research.sun.com/self>
  - C++ has the "this pointer"
  - Java has the "this reference"

4/19/04

10

## Dynamic Scoping

- One way to implement object-oriented semantics is to use dynamic scoping and a self reference

- this is not exactly how it is done in C++ and Java
- class NAT = record

```
var NUM:newint;
proc SUCC = self.NUM := @self.NUM+1;
proc PLUSTWO = call self.SUCC;
                call self.SUCC
```

```
end
```

```
class INT = inherits NAT with record
```

```
var ISNEG:newbool;
proc SUCC = if !self.ISNEG
              then self.NUM := @self.NUM+1;
              else self.NUM := @self.NUM-1;
              if @self.NUM=0
                then self.ISNEG:=false
              fi fi
```

```
end
```

4/19/04

11

## Dynamic Scoping

- We can now introduce an object declaration, which implicitly resolves the self references

- object N:NAT, object I:INT**

- N is a record that is instantiated with all references to self resolved to N (the record value), resulting in a kind of recursive variable declaration:

```
rec-var N: record var NUM:newint;
              proc SUCC = N.NUM:=@N.NUM+1;
              proc PLUSTWO = call N.SUCC;
                          call N.SUCC
```

```
end
```

- this technique essentially binds the record value in-place in of the references to self, so that for a given instance of a class, all local variables and methods are referenced (addressable) relative to the dynamic value of 'self'

4/19/04

12

## This Pointer in C++

- used to dynamically bind an object's state (instance variables) to the methods for the class of that object at run-time
  - if an object 'a' is of type X, then a.m() is short-hand for X::m(X\* const this=&x). Similar, p->m() is shorthand for X::m(X\* const this=p)
  - what is the type of "this" in C++?
    - it varies depending on how the method is defined
    - X\* const for non-const methods
    - const X\* const for const methods
  - static member functions do not have a "this" --- why?
  - in C++, the this pointer's r-value is one of:
    - a global region address - object allocated global/static
    - a stack region address - objects allocated auto
    - a heap region address - object allocated with new

4/19/04

13

## This Pointer in C++

- the this pointer does not occupy any space in an object. its storage is external to an object
- it is illegal to assign to the this pointer in a method
  - early versions of C++ allowed you to change this inside of a method!
- but you can use it in expressions where it is treated as an r-value only
  - `foo(int x) { int y = x + this->x; }`
  - `if (this != &that) ...`

4/19/04

14

## This Pointer in C++

- The this pointer is also used to resolve dynamic method dispatching
  - C++ generates a special data structure called a vtable (virtual function dispatch table) for every object that contains at least one virtual function
  - a vtable pointer is cached in each object, and is accessed using the this pointer for the object
    - so a virtual function call requires three pointer dereferences to invoke a virtual method
      - dereference the this pointer to obtain the object
      - dereference the vtable pointer to obtain the vtable
      - index into the vtable and dereference a function pointer to invoke the function

## Homework

- Continue reading chapter 5 (last chapter!)
- Read the following paper from the course web site
  - Uniprocessor Garbage Collection Techniques, by Paul R. Wilson
- Keep putting effort into your course project
  - 3 weeks left to completion date