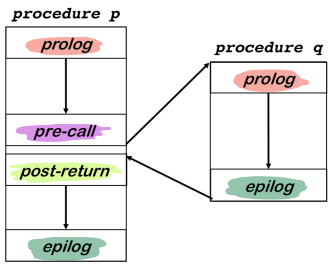
Run-Time Environments

- See Activation Record (AR) in the List of Class Materials.

Procedure Linkages

É preciso verificar registos, variáveis (incluindo parâmetros), Allocated space, Access links e addresses Antes e depois de uma chamada a um método. (Tal como foi dado em AC com assembly)

Standard procedure linkage



Procedure has
- standard **prolog**
- standard **epilog**

Each call involves a
- **pre-call** sequence
- **post-return** sequence

These are completely predictable from the Call Site as they depend on the number & type of the actual parameters

Pre-Call Sequence
- Sets up Callee's basic AR
- Helps preserve its own environment

The Details
- Allocate Space for the Callee's AR
  - except space for local variables
- Evaluates each Parameter & Stores Value or Address
- Saves Return Address, caller's ARP into Callee's AR
- If Access Links are used
  - Find appropriate lexical ancestor & copy into Callee's **AR**
- Save any Caller-save Registers
  - Save into space in Caller's AR
- Jump to Address of Callee's prolog code

Post-Return Sequence
- Finish restoring Caller's environment
- Place any value back where it belongs

The Details
- Copy return value from Callee's AR, if necessary
- Free the Callee's AR
- Restore any caller-save registers
- Restore any call-by-reference parameters to registers, if needed
  - Also copy back call-by-value/result parameters
- Continue execution after the call

Prolog and Epilog Code

Prolog Code
- Finish setting up the Callee's environment
- Preserve parts of the Caller's environment that will be disturbed

The Details
- Preserve any Callee-save registers
- If *Display* is being used
  - Save display entry for current lexical level
  - Store current ARP into display for current lexical level
- Allocate Space for Local Data
  - Easiest scenario is to extend the AR
- Find any Static Data areas referenced in the Callee
- Handle any Local Variable Initializations

With heap allocated AR, may need to use a separate heap object for local variables

Epilog Code
- Wind up the business of the Callee
- Start restoring the Caller's Environment

If ARs are stack allocated, this may not be necessary. (Caller can reset stack top to its pre-call value.)

The Details
- Store Return Value? No, this happens on the return statement
- Restore Callee-save Registers
- Free space for Local Data, if necessary (on the Heap)
- Load Return Address from AR
- Restore caller's ARP
- Jump to the Return Address

Esta secção tem como objetivo garantir que as alterações dos dados são feitas em conformidade, para garantir consistência entre pontos onde os processos são chamados e retornados.

Caller-Saved VS Callee-Saved Registers

- **Caller-saved registers** (**volatile** registers, or **call-clobbered**)
  - Hold temporary quantities that need not be preserved across Calls.
  - Caller's responsibility to push these registers onto the stack or copy them somewhere else *if* it wants to restore this value after the call.
  - Expected callee to ***destroy*** temporary values in these registers…
- **Callee-saved registers** (**non-volatile** registers, or **call-preserved**)
  - Used to hold long-lived values that should be preserved across Calls.
  - Callee's responsibility to push then onto the stack or copy them somewhere else *if* it wants to restore this value after the call.
  - Expected callee to ***preserve*** (not destroy) temporary values in these registers…

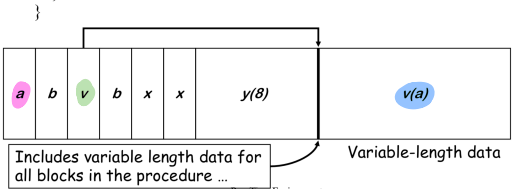
See Assembly Slides to see how it works in practice. (just like AC classes)

Variable-Length Data

```
B0: {
      int a, b
      … assign value to a
B1:   {
        int v(a), b, x
B2:     {
          int x, y(8)
          ….
        }
      }
    }
```

Arrays

- → If size is fixed at compile time, store in fixed-length data area
- → If size is variable, store **descriptor** in fixed length area, with pointer to variable length area
- → Variable-length data area is assigned at the end of the fixed length area for block in which it is allocated

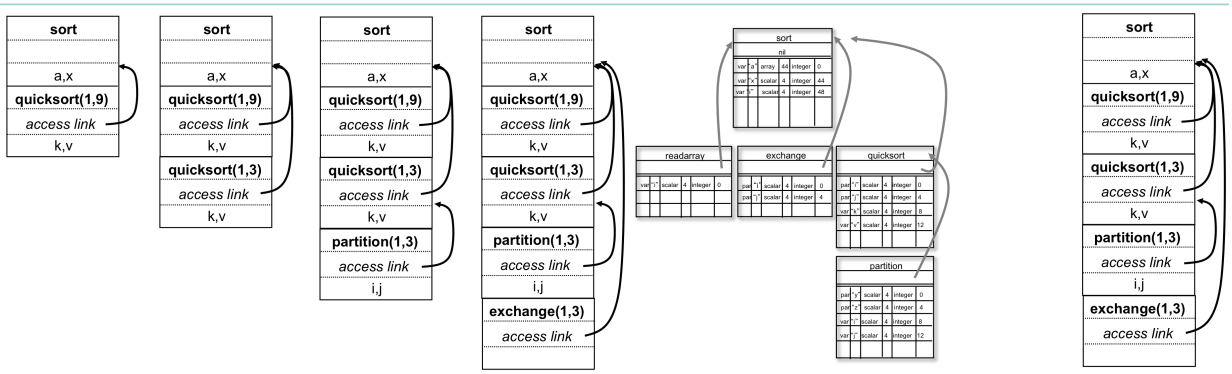


Translating Local Names

How does the compiler represent a specific instance of x?

- Name is translated into a ***static coordinate***
  - < *level*,*offset* > pair
  - "*level*" is lexical nesting level of the procedure
  - "*offset*" is *unique* within that scope
- Subsequent code will use the static coordinate to generate addresses and references
- "*level*" is a function of the table in which x is found
  - Stored in the entry for each x
- "*offset*" must be assigned and stored in the Symbol Table
  - Assigned at Compile time
  - Known at Compile time
  - Used to Generate code that executes at run-time

Lexical Scope with Nested Procedures



See Display Algorithm in the slides 53-62
↳ how to generate code for body

First Summary

- What Have We Learned?
  - AR is a Run-time Structure to hold State regarding the Execution of a Procedure
  - AR can be allocated in Static, Stack or even Heap
  - Links allow Call-Return and Access to Non-local Variables
  - Symbol-Table plays Important Role

Important

descendant order of efficiency!

- Linkage Conventions
  - Saving Context before Call and restoring after Call
  - Need to understand how to generate code for body

Object-Oriented Languages

Mapping "message" or names to methods

- Static mapping, known at compile-time (Java, C++)
  - Fixed offsets & indirect calls
- Dynamic mapping, unknown until run-time (Smalltalk)
  - Look up name in class' table of methods

Want uniform placement of standard services *(**NEW, PRINT**, …)*

This is really a Data-Structures Problem

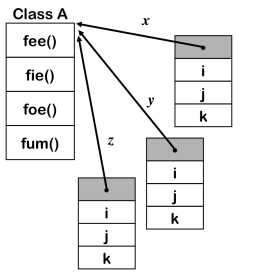
- Build a Table of Function Pointers
- Use a Standard Invocation Sequence

Class Structure

Method Code:

```
void A:fee(){
  ...
  self->i = 0;
  ...
}
```

```
t0 = &self + 4;
*t0 = 0;
```



objects of class A

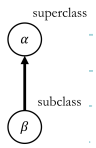
Method Accesses Object Data as Offsets from the **self** Reference

Some Terminology

1. **Object** An object is an abstraction with one or more members. Those members can be data items, code that manipulates those data items, or other objects. Each object has internal state—data whose lifetimes match the object's lifetime.
2. **Class** A class is a collection of objects with the same abstract structure and characteristics. A class defines the set of data members in each instance of the class and defines the code members (methods) that are local to that class. Some methods are public, or externally visible, others are private, or invisible outside the class.
3. **Inheritance** Inheritance refers to a relationship among classes that defines a partial order on the name scopes of classes. Each class may have a superclass from which it inherits both code and data members. If a is the superclass of b, b is a subclass of a. Some languages allow a class to have multiple superclasses.
4. **Receiver** Methods are invoked relative to some object, called the method's receiver. The receiver is known by a designated name, such as *this* or *self*, inside the method.

Inheritance (Simple)

- If α is a superclass of β, then β is a subclass of α and any method defined in α must operate correctly on an object of class β, if it is visible in β.
- The converse is not true; a method declared in class β cannot be applied to an object of its superclass α, as the method from β may need fields present in an object of class β that are absent from an object of class α



β derives from α

Inheritance Hierarchy

Two distinct Implementation Philosophies

Static Class Structure	Dynamic Class Structure
• Can map name to code at compile time	• Cannot map name to code at compile time
• Leads to 1-level jump vector	• Multiple jump vector (1/class)
• Copy superclass methods	• Must search for method
• Fixed offsets & indirect calls	• Run-time lookups caching
• Less flexible & expressive	• Much more expensive to run

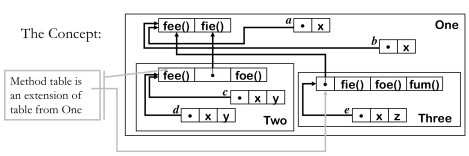
Impact on name space

- Method can see instance variables of self, class, & superclasses
- Many different levels where a value can reside

In essence, OOL differs from ALL in the shape of its name space ***AND*** in the mechanism used to bind names to implementations

To simplify object creation, we allow a class to inherit methods from an ancestor, or *superclass*. The descendant is called the *subclass* of its ancestor.

The Concept:



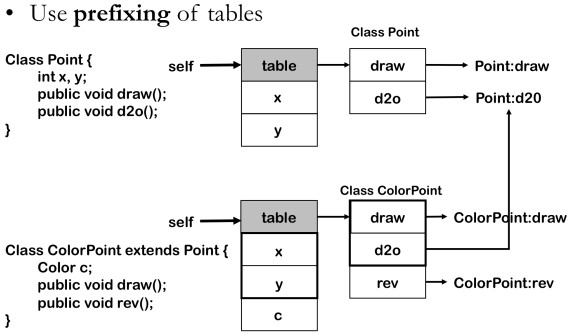
Method table is an extension of table from One

Principle:

- B *subclass* A ⇒ d ∈ B can be used wherever a ∈ A is expected
  - B has all the methods defined in its super class (in this case class A)
  - B may override a method definition from A
- Subclass provides all the interfaces of superclass

Single Inheritance and Dynamic Dispatch

- Use **prefixing** of tables



Multiple Inheritance

The Idea:

- Allow more flexible sharing of methods & attributes
- Relax the inclusion requirement
  If B is a subclass of A, it need not implement all of A's methods
- Need a linguistic mechanism for specifying partial inheritance

Problems when C inherits from both A & B

- C's method table can extend A or B, but not both
  - Layout of an object record for C becomes tricky
- Other classes, say D, can inherit from C & B
  - Adjustments to offsets become complex
- Say, both A & B might provide fum() — which is seen in C ?
  - C++ produces a "syntax error" when fum() is used

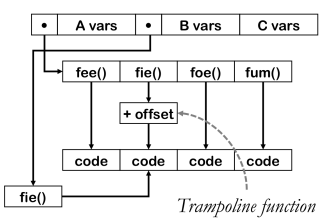
Casting with Multiple Inheritance

- Usage as Point:
  - No extra action (prefixing does everything)
- Usage as CThing:
  - Increment self by 12
- Usage as CPoint:
  - Lay out data for CThing at self + 12
  - When calling the rev method
    - Call in table points to a *trampoline* function that adds 12 to self, then calls rev
    - Ensures that rev, which assumes that self points to a CThing data area, gets the right data (at the correct layout offsets…)

classes from here

Another Example

Assume that C inherits fee() from A, fie() from B, & defines both foe() and fum()



This implementation

- Uses *trampoline* functions
- Optimizes well with inlining
- Adds overhead where needed (Zero offsets go away)
- Folds inheritance into data structure, rather than linkage

Assumes static class structure

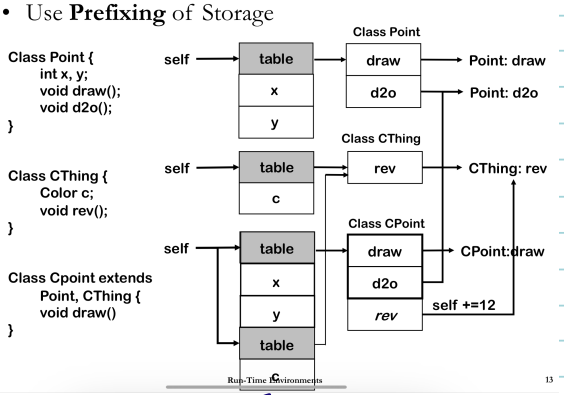
For dynamic, why not rebuild on a change in structure?

Second Summary

- Support for Object-Oriented Languages:
  - Mostly focus on Message Dispatching
  - Finding the "correct" function in the Class Hierarchy
  - Adjusting Object Layout and Class References
- An Invocation Stack is still needed....

Example

- Use **Prefixing** of Storage



So, what can an Executing Method see?

- The Object's own Attributes & Private Variables
- The Attributes & Private Variables of Classes that define it
  - *May be many such classes, in many combinations*
  - *Class variables are visible to methods that inherit the class*
- Object defined in the Global Name Space (or scope)
  - Objects may contain other objects
- Objects that contain their definition
  - *A class as an instance variable of another class, ...*

An Executing Method might reference any of these

Making this work requires compile-time elaboration for static case and run-time elaboration for dynamic case

Making it run quickly takes care, planning, and trickery…