



COMP 181

Lecture 12
Procedures and methods

October 24, 2005





Prelude

$$\frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

- What language is this?


MULTIPLY B BY B GIVING B-SQUARED.
 MULTIPLY 4 BY A GIVING FOUR-A.
 MULTIPLY FOUR-A BY C GIVING FOUR-A-C.
 SUBTRACT FOUR-A-C FROM B-SQUARED GIVING RESULT-1.
 COMPUTE RESULT-2 = RESULT-1 * .5.
 SUBTRACT B FROM RESULT-2 GIVING NUMERATOR.
 MULTIPLY 2 BY A GIVING DENOMINATOR.
 DIVIDE NUMERATOR BY DENOMINATOR GIVING X.
- How widely used is COBOL?
 - All languages combined: 300 billion LOC
 - COBOL accounts for 240 billion



2

Project

- Semantic checks
 - Type checking
 - Connecting identifiers and declarations
 - Usage rules
- Central process: walking the AST
 - Visit nodes, check configuration
 - Descend to children
- How to best represent this?




3

Heterogeneous data structures

- Simple example: shapes
 - Abstract type "shape"
 - Concrete subtypes: square, circle, triangle
- C:


```
typedef struct Shape {
    int kind; /* 0=square, 1=circle, 2=triangle */
    union {
        struct Square { . . . } square;
        struct Circle { . . . } circle;
        struct Triangle { . . . } triangle;
    } info;
} Shape, * ShapePtr;
```




4

Labeled unions

- Visiting a series of shapes



```
Shape * all_shapes[50];
float area = 0.0;
for (int i = 0; i < 50; i++) {
    Shape * cur_shape = all_shapes[i];
    switch (cur_shape->kind) {
        case 0: area += cur_shape->stuff.square.side *
                    cur_shape->stuff.square.side;
                break;
        case 1: . . .
        case 2: . . .
    }
}
```



5

Labeled unions

- What's good?
 - Code for "area" all in one place
 - Probably fast
- What's bad?
 - Have to make sure to get kinds right (Could use enum to help)
 - No control over access to fields
 - New shapes: fix all the switch statements



6

Objects

- Common superclass
 - Common functionality
 - Required functionality

```
class Shape {
    abstract float area();
}
class Square extends Shape {
    float side;
    float area() { return side * side; }
}
class Circle extends Shape { . . . }
class Triangle extends Shape { . . . }
```



Objects

- What's good
 - Encapsulation – all fields protected
 - No type errors
 - Easy to add new shapes
- What's bad
 - How do we add new functions?



Visitor

- Method for each kind

```
class Visitor {
    void visitSquare(Square sq) { }
    void visitCircle(Circle ci) { }
    void visitTriangle(Triangle tr) { }
}
```

- Idea:
 - Describe what to do with each kind of object
 - Object calls its visit class



Visitor

- Implement subclass:

```
class areaVisitor extends Visitor {
    float area = 0.0;
    void visitSquare(Square sq) {
        area += sq.side() * sq.side();
    }
    void visitCircle(Circle ci) {
        area += pi * ci.radius() * ci.radius();
    }
    void visitTriangle(Triangle tr) { . . . }
}
```

- Note:
 - May need to carry data in the visitor



Supporting visitors

- In the shapes classes

```
class Shape {
    abstract float area();
    void accept(Visitor vis) {
        System.out.println("Unimplemented");
    }
}
class Square extends Shape {
    float side;
    void accept(Visitor vis) {
        vis.visitSquare(this);
    }
}
```



Using a visitor

- Three parts
 - Create visitor instance
 - Call "accept" method on each shape
 - Read out the result at end

```
Shape[] all_shapes;
Visitor vis = new areaVisitor();
for (int i = 0; i < 50; i++) {
    Shape cur_shape = all_shapes[i];
    cur_shape.accept(vis);
}
float result = vis.area;
```



Visitor

- What's good
 - Easy to add new "passes"
 - All code is in one place
- What's bad
 - New shapes: have to modify all visitors



Today

- Review
 - Nested scopes
 - Symbol tables
- Implementing procedure linkage
- Object-oriented languages



Scopes

- Lexical
 - Scope can be described as a contiguous region of the program text
- Nested
 - Open new scopes within other scopes
 - Fresh names within each scope



Compiler responsibilities

- Name resolution
 - Match uses of names to declarations
 - Which "x" is the programmer talking about?
- Storage allocation
 - Set up storage for each variable
 - Generate appropriate address computations
 - At run-time, where is the value of "x"?



Storage

- What is appropriate location for a variable?
- Naïve implementation
 - *Set aside a big block of memory, assign each variable in the program one location in block*
- What's wrong with this?
 - Is it space efficient?
 - Where does it fail?



Storage

- Observation:
 - *The scope of a variable implies lifetime*
- How does this help?
 - Many variables not in use at same time
 - *They can share space*
 - Some variables can have multiple instances at the same time (recursion)
 - *Allocation must be dynamic*



Example

```

int global;
static int local_global;
extern int ext_global;
int foo(int x, float y)
{
  int a, b;
  for (int i = 0; i < 10; i++) {
    float temp = foo(x-1, y);
  }
  while (y < 3.141) {
    int temp = ...;
  }
  return ...;
}

```

Whole program

```

global,
local_global
foo
  foo()
    x, y, a, b
  for
    i, temp
  while
    temp

```

Tufts University Computer Science 19

Example

Whole program

```

global,
local_global
foo
  foo()
    x, y, a, b
  for
    i, temp
  while
    temp

```

Fixed place in address space of program

Fixed offsets relative to activation record of procedure

Tufts University Computer Science 20

Example

Base address

```

foo()
  x, y, a, b
  for
    i, temp
  while
    temp

```

x at offset 0
y at offset +4
a at offset +12
b at offset +16

i at offset +20
temp at offset +24

Tufts University Computer Science 21

Example

- Code for `foo()`
 - Assume base address in common location
 - Stack pointer – “sp” – often a register
 - Access to variable is base+offset
 - At a call-site, move base address – “push”

Statements like

```

b = a + x;

```

become

```

*(sp+16) = *(sp+12) + *(sp)

```

Tufts University Computer Science 22

At run-time...

Call to `foo()`

Recursive call to `foo()`

Stack grows down

Tufts University Computer Science 23

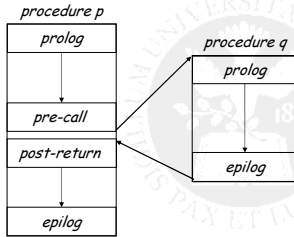
Procedure linkages

- Procedure call implementation
 - What does the compiler have to work with?
 - Load, store
 - Jump
 - Where are the caller and callee?
 - Different compilation units
 - Library code
- Procedure call is a system-wide protocol
 - Orchestrated by compiler
 - Add code to caller and callee

Tufts University Computer Science 24

Procedure Linkages

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 \Rightarrow depend on the number & type of the actual parameters



Procedure linkages

- **Pre-call** sequence
 - Sets up callee's basic AR
 - Helps preserve its own environment
- The details
 - Allocate space for the callee's AR : **(excluding local vars)**
 - Evaluates each parameter & stores value or address
 - Saves return address, caller's ARP into callee's AR
 - Save any caller-save registers **(save in caller's AR)**
 - Jump to address of callee's prolog code



Procedure linkages

- **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
 - Also copy back call-by-value/result parameters
 - Continue execution after the call



Procedure linkages

- **Prolog code**
 - Finish setting up callee's environment
 - Preserve parts of caller's environment that will be disturbed
- The details
 - Preserve any callee-save registers
 - Allocate space for local variables
 - 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



Procedure linkages

- **Epilog Code**
 - Wind up the business of the callee
 - Start restoring the caller's environment
- The details
 - Store return value?
 - Some implementations do this on the return statement
 - Others have return assign it & epilog store it into caller's AR
 - 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

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



Back to activation records

- Stored on the stack
 - Easy to extend — simply bump top of stack pointer
 - Caller & callee share responsibility
 - Caller can push params, space for registers, return value slot, return address, addressability info, & its own ARP
 - Callee can push space for local variables
- Stored on the heap
 - Hard to extend
 - Caller passes everything it can in registers
 - Callee allocates AR & stores register contents into it
 - Extra parameters stored in caller's AR !
- Static is easy



Object-Oriented Languages

- What is an OOL?

How is it different from an ALL? (Algo-like language)

- Data-centric view
 - Objects communicating by “messages” – *methods*
- Polymorphism
 - Meaning of message depends on receiver
- Inheritance
- What was first OOL?
- Term is almost meaningless today



Object-Oriented Languages

An object is an abstract data type that encapsulates data, operations and internal state behind a simple, consistent interface.

The Concept:



Issues for compilation:

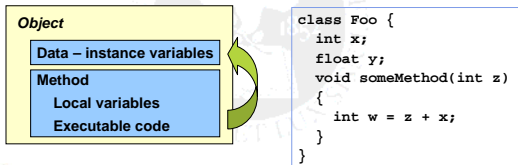
- Each object needs local storage for its attributes
- Access control – notion of privacy
- More complex linkage convention – dynamic aspect
- Object's internal state leads to complex behavior



Object structure

- Idea:

- Natural extension of scope
- Methods can see data in the object, in addition to usual local scopes



Method abstraction

- Object-oriented name spaces

- Local storage in objects (beyond attributes)
- Some storage associated with methods
 - Local values inside a method
 - Static values with lifetimes beyond methods
- Methods shared among multiple objects

- Classes

- Objects with the same state are grouped into a class
 - Same attributes, instance variables, & methods
 - Class variables are static, shared among all objects of same class
- Allows programmer to write it down once
- Allows code reuse at both source & implementation level



Implementation

So, what can an executing method see?

- The object's own attributes & private variables
 - Smalltalk terminology: *instance variables*
 - The attributes & private variables of the class that defines it
 - Smalltalk terminology: *class variables*
 - Any object defined in the global name space (or scope)
 - Objects may contain other objects (!?)
- ⇒ An executing method might reference any of these

A final twist:

- Most OOLs support a hierarchical notion of inheritance
- Some OOLs support multiple inheritance



Java Name Space

Given a method M for an object O in a class C, M can see:

- Local variables declared within M (*lexical scoping*)
- All instance variables and class variables of the class C
- All public and protected variables of any superclass of C
- Classes in the same package or in an imported package
 - public class variables and instance variables of imported classes
 - package class and instance variables in the package containing C
- Class declarations can be nested



Java Symbol Tables

To compile code in method M for an object O within a class C, the compiler needs:

- Lexically scoped symbol table for block and class nesting
 - Just like ALL — inner declarations hide outer declarations
- Chain of symbol tables for inheritance
 - Need mechanism to find the class and instance variables of all superclasses
- Symbol tables for all global classes (package scope)
 - Entries for all members with visibility
 - Need to construct symbol table for imported packages



Java Symbol Tables

To find the address associated with a variable reference in method M for an object O within a class C, the compiler must

- For an unqualified use (i.e., x):
 - Search the scoped symbol table for the current method
 - Search the chain of symbol tables for the class hierarchy
 - Search global symbol table (current package and imported)
 - In each case check access control attribute of x
- For a qualified use (i.e.: Q.x):
 - Search stack-structured global symbol table for Q
 - Check access control attribute of x



Implementation

Two critical issues in OOL implementation:

- Object representation
- Mapping a **method invocation** name to a **method implementation**

These both are intimately related to the name space

Object Representation

- Private storage for attributes & instance variables
- Need consistent, fast access
- Provision for initialization in NEW



OOL Storage Layout (Java)

Class variables

- Static class storage accessible by global name (*class C*)
 - Accessible via linkage symbol &_C
 - Nested classes are handled like blocks in ALLs
 - Method code put at fixed offset from start of class area

Object Representation

- Object storage is heap allocated
 - Fields at fixed offsets from start of object storage
- Methods
 - Code for methods is stored with the class
 - Methods accessed by offsets from code vector
 - Method local storage in object (no calls) or on stack



Dealing with Single Inheritance

- Use **prefixing** of storage

```
Class Point {
  int x, y;
}
```

```
Class ColorPoint extends Point {
  Color c;
}
```

- What's nice about this?
 - Polymorphism "for free"
 - Code generated for Point will work on ColorPoint



Implementation

Mapping message 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

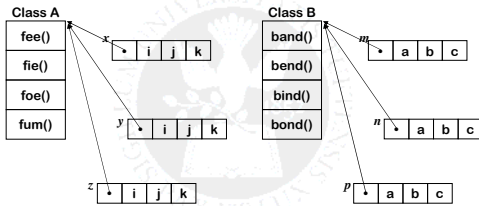
This is really a data-structures problem

- Build a table of function pointers
- Use a standard invocation sequence



Implementation

With static, compile-time mapped classes

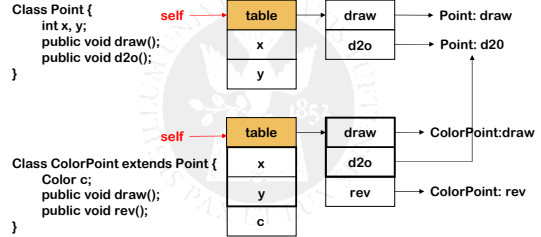


Message dispatch becomes an indirect call through a function table



Single Inheritance and Dynamic Dispatch

- Use **prefixing** of tables



The Inheritance Hierarchy

- Two distinct philosophies

Static class structure

- Can map name to code at compile time
- Leads to 1-level jump vector
- Copy superclass methods
- Fixed offsets & indirect calls
- Less flexible & expressive

Dynamic class structure

- Cannot map name to code at compile time
- Multiple jump vector (t/class)
- Must search for method
- Run-time lookups caching
- Much more expensive to run

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



Next time...

- I'll return homework
- Review for "midterm"
- Intermediate representations

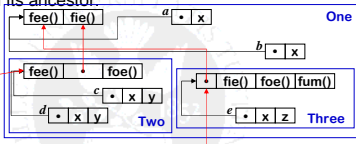


The Inheritance Hierarchy

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

The Concept:

Method table is an extension of table from One



Principle

- $B \text{ subclass } A \Rightarrow d \in B$ can be used wherever $e \in A$ is expected
 - B may override a method definition from A
- Subclass provides all the interfaces of superclass,



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
- Both A & B might provide $\text{fum}()$ — which is seen in C ?
 - $C++$ produces a "syntax error" when $\text{fum}()$ is used

Need a better way to say "inherit!"



Multiple Inheritance Example

- Use **prefixing** of storage

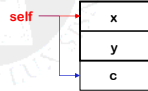
```
Class Point {
  int x, y;
}
```



```
Class ColoredThing {
  Color c;
}
```



```
Class ColorPoint extends
  Point, ColoredThing {
}
```



Does casting work properly?



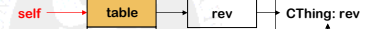
Multiple Inheritance Example

- Use **prefixing** of storage

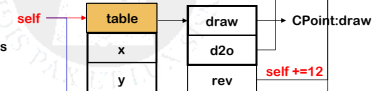
```
Class Point {
  int x, y;
  void draw();
  void d2o();
}
```



```
Class CThing {
  Color c;
  void rev();
}
```



```
Class CPoint extends
  Point, CThing {
  void draw()
}
```



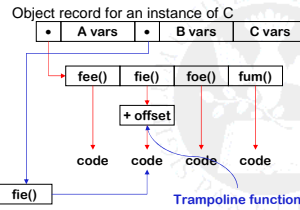
Casting with Multiple Inheritance

- 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 + 16**
 - When calling **rev**
 - 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



Multiple Inheritance (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?

