Runtime Support for OOLs
Object Records, Code Vectors, Inheritance
Comp 412

Copyright 2018, Keith D. Cooper & Linda Torczon, all rights reserved.
Students enrolled in Comp 412 at Rice University have explicit permission to make copies of these materials for their personal use.
Faculty from other educational institutions may use these materials for nonprofit educational purposes, provided this copyright notice is preserved.

Chapter 6 & 7 in EaC2e
Support for Name Spaces

In an ALL, the compiler needs
• Compile-time mechanism for name resolution
• Runtime mechanism to compute an address from a name
Compiler must emit code that builds & maintains the runtime structures for addressability

In an OOL, the compiler needs
• Compile-time mechanism for name resolution
• Runtime mechanism to compute an address from a name
Compiler must emit code that builds & maintains the runtime structures for addressability

This lecture focuses on runtime support for OOLs.

COMP 412, Fall 2018
Runtime Support for OOLs

Where are we?

• We have seen:
  – Symbol tables for lexically-scoped ALLs
  – Symbol tables & object tables for OOLs
  – Linkage conventions for procedure calls (in ALLs and OOLs)
  – Access links & displays to provide runtime support for lexical scopes

• ALL addressability relies on the ARP and name mangled labels
  – An OOL adds addresses that are relative to the receiving object

• Today: runtime structures to support addressability in an OOL
  – In essence, given an object fee, how do we find fee.method() and fee.member
To compile method M of object O in class C, the compiler needs:

1. Lexically scoped symbol table for the current block and its surrounding scopes in M
   - Just like an ALL — inner declarations hide outer declarations

2. Chain of symbol tables for inheritance
   - Class C and all of its superclasses
   - Need to find methods and instance variables in any superclass

3. Symbol tables for all global classes (package scope)
   - Entries for all members with visibility
   - Need to construct symbol tables for imported packages and link them into the structure in appropriate places

Three sets of tables are needed for name resolution. In an ALL, we could combine 1 & 3 for a single unified set of tables. In Java, we need to split them so that the compiler can check the inheritance hierarchy between the lexical hierarchy & the global name space.
From the lecture on naming

Compile-time Structures for OOLs — Java

Conceptually

Search Order: lexical hierarchy, class hierarchy, global scope

Note the “sheaf of tables” model.
The classic example for discussing inheritance. We will use & extend it.

Class Point {
    public int x, y;
    public void draw();
}
Class ColorPoint extends Point {
    Color c;
    public void draw() {...}
    public void test() { y = x; draw(); }
}
Class C {
    int x, y;
    public void m()
    {
        Point p = new ColorPoint();
        y = p.x;
        p.draw();
    }
}

From the lecture on naming
Runtime Structures for OOLs

Object lifetimes are independent

- Each object needs an object record (OR) to hold its state
  - Independent allocation and deallocation
    - *Think “heap”*
  - We will talk of OR pointers, much like AR pointers
    - *We won’t call them ORPs*

- Classes are objects, too
  - ORs of classes instantiate the class hierarchy

Object Records

- Distinct static storage for data members
- Need fast, consistent access
  - Known constant offsets from OR pointer
- Linguistic provision for initialization

This discussion is generic rather than Java-specific.

The Concept

<table>
<thead>
<tr>
<th>fee()</th>
<th>fie()</th>
<th>foe()</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>8</td>
<td>12</td>
</tr>
</tbody>
</table>

The Heap

address polynomial for a vector, similar to fields in an activation record.
Runtime Structures for OOLs

From Concept to Implementation

- Members of a class should share code members
- Replication would waste space
- Replication would be a nightmare

The Concept

The Implementation

- Single copy of the code for each method
- Class has a vector of “code pointers”
- Place code vector @ fixed offset in class’ OR

Could place code pointer at offset 0 to eliminate an add...
From Concept to Implementation

- Members of a class should share code members
- Replication would waste space
- Replication would be a nightmare

The Implementation

- Single copy of the code for each method
- Move code pointer into object for faster access
  - One more slot in the OR
  - One less indirection in each call
The Basics of Inheritance

Inheritance is the primary mechanism for code reuse in OOLs

• Class \( y \) derives definitions of its members from class \( x \)
  – \( y \) is a subclass of \( x \) and \( x \) is a superclass of \( y \)
  – Can have multi-level inheritance

• Subclass has all members of superclass
  – Includes transitive superclasses (\( super\text{-}super \), \( ... \))

• Subclass can add new members

• Subclass can override inherited definitions
  – ColorPoint inherits \( x \) & \( y \) from Point, adds \( m \), and overrides \( draw() \)

Some languages allow a class \( x \) to have two or more superclasses

• This feature is called multiple inheritance

• Can introduce conceptual problems with name resolution

• Languages use different rules to disambiguate these situations

COMP 412, Fall 2018
The class hierarchy must be represented with runtime objects

Each class is, itself, an object

- Class, superclass & code vector pointers are data members of class Class
- `p.draw()` is found as `p.class -> Point.code + offset -> p.draw`\(^1\)
- Hierarchy must be built (statically, at initialization, or at class load)

\(^1\) Code vector could exist in each object, which would make method dispatch faster at the cost of more space in each object. Much of the engineering on OOLs was done in small memory days.
OR Layout

Inheritance has a strong impact on OR Layout

- **OR** needs slots for each declared member, all the way up the class hierarchy chain ($class$, $superclass$, $super-superclass$, ...)
  - Each object needs an instance variable that points to its class
  - Each class needs an instance variable that points to its superclass
- Can use **prefixing** of storage to lay out the **OR**

Back to Our Java Example — Class Point

Class Point {
    public int x, y;
    ...
}

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

What happens if we cast a **ColorPoint** to a **Point**?
A **Point** to a **ColorPoint**?

Take the word **extends** literally.
Open World versus Closed World

Compile-time prefixing assumes that the class structure is known when layout is performed. Two common cases occur in OO language design.

Closed-World Assumption (Compile time)
- Class structure is known and closed prior to runtime
- ORs can be laid out in the compiler and/or the linker
- Method calls can be direct, unless runtime resolution is specified

Open-World Assumption (Interpreter or JIT)
- Class structure can change at runtime
  - Import or create classes at runtime
- ORs layout might change at runtime
  - Walk class hierarchy at first instantiation
  - Need the compile-time data structures at runtime

C++ has a closed class structure.
Java as an open class structure.
Wait! Explain “Virtual Function”

A “virtual” function is explicitly declared to resolve at runtime

Why does this matter?

• Assume: Point is ColorPoint’s superclass & both classes provide `draw()`
  – Program creates a Point `q`
  – Program assigns both Points & ColorPoints to `q`
  – Program invokes `q.draw()`
• Which `draw()` is invoked?

It depends on when the function name is resolved, or “bound”

• Compile-time binding would use `q’s` type and invoke Point’s `draw()`
• Runtime binding would use the `p’s` type & invoke ColorPoint’s draw
• In C++, the “virtual” keyword forces runtime resolution of that name
  – Virtual declaration in superclass holds for all subclasses
  – Forces a runtime search through the object’s class pointer

Languages use combinations of virtual function resolution and abstract functions to implement abstraction mechanisms such as abstract classes and interfaces.

Too many uses of “abstract” ...
How does the running program find the code for a given method invocation?

Closed Class Structure

- Mapping of names to methods is static and known (C++)
  - Fixed offsets & indirect calls \((\text{class} \rightarrow \text{code vector} \rightarrow \text{function pointer})\)
- Virtual functions force runtime resolution

\[
\text{cp finds draw at offset 0 in ColorPoint's code vector}
\]
Runtime Resolution for Code Members

How does the running program find the code for a given method invocation?

Closed Class Structure

• Mapping of names to methods is static and known (C++)
  – Fixed offsets & indirect calls \( \text{class} \rightarrow \text{code vector} \rightarrow \text{function pointer} \)

• Virtual functions force runtime resolution

If ColorPoint inherited `draw` from Point, its code vector would refer to Point’s `draw`.

\[ p \text{ is a Point} \]
\[ \text{cp is a ColorPoint} \]

\[ \text{cp finds draw at offset 0 in ColorPoint’s code vector} \]
Runtime Resolution for Code Members

How does the running program find the code for a given method invocation?

Open Class Structure

- Dynamic mapping, unknown until runtime
- In restricted cases, can build complete code vectors at each change
  - Must lookup offsets, but can then reference into vector at a fixed offset

<table>
<thead>
<tr>
<th>Name</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>draw</td>
<td>0</td>
</tr>
<tr>
<td>test</td>
<td>4</td>
</tr>
</tbody>
</table>

Superclass methods found by walking hierarchy

*cp* finds *draw* at offset 0 in ColorPoint’s code vector
Runtime Resolution for Code Members

How does the running program find the code for a given method invocation?

Open Class Structure

- Dynamic mapping, unknown until runtime
- In general case, need to search the runtime representation of hierarchy
  - Lookup by textual name in class’ table of methods \((\text{truly slow!})\)
Runtime Resolution for Code Members

How does the running program find the code for a given method invocation?

Open Class Structure

• Dynamic mapping, unknown until runtime
• In general case, need runtime representation of hierarchy
  — Lookup by textual name in class’ table of methods (truly slow!)

If ColorPoint inherited draw from Point, its code vector would lack a draw. It would find draw at offset 0 in Point’s code vector.

Smalltalk 80
Runtime Resolution for Code Members

How does the running program find the code for a given method invocation?

Open Class Structure
- Dynamic mapping, unknown until runtime
- In general case, need runtime representation of hierarchy
  - Lookup by textual name in class’ table of methods (truly slow!)

And, in an open class structure, the “code vector in each OR” design would create many, many updates when a change in the class structure changes the code vector(s). (efficiency?)

General problem: updating ORs of instantiated objects when their class changes (language design issue)

Superclass methods found by walking hierarchy
Closed World versus Open World

**Closed Class Structure**
- Compile-time or link-time layout (**ORs**, classes, code vectors)
- If known at compile time, can generate static code
  - No code vector, no indirection, efficient implementation
  - Locate code the old fashioned (& fast) way
- C++ virtual functions use runtime resolution, as if open world

**Open Class Structure**
- Cannot lay out **ORs** until hierarchy is known
- With infrequent change, can change code vectors at each change
  - Single-level code vectors, fixed offsets, indirect calls
  - Instantiated objects of changed classes are still a problem
- With frequent change, may need full runtime hierarchy
  - Search for method in class hierarchy (w/tag) & cache result
  - Much more expensive, on each call or on first call

*Smalltalk 80*
Method Calls

Given the runtime structure, how does a call work?

• Compiled code does not contain the callee’s address
• Reference is relative to receiver
  – Follow the chain: OR → class → code vector (or OR → code vector)
  – Call overhead is higher, in the best case
• Rest of pre-call, post-return, prolog, & epilog are as in an ALL

In the general case, may need dynamic dispatch

• Map code member to a search key
• Perform runtime search through hierarchy
  – This process is expensive, potentially very expensive
• Use a “method cache” to speed the search
  – Cache holds <search key, class, method pointer>

General case: Virtual functions where choice of method depends on runtime type of actual receiver — bound “late” or “at runtime.” In Smalltalk 80, all method dispatch worked this way.
Method Calls

Improvements are possible in special cases

• If class has no subclasses, can generate direct calls
  – Class structure must be static or class must be `FINAL`

• If class structure is static
  – Can generate complete method table for each class
    → Use prefixed object records and complete code vectors
  – Indirection through the class pointer or pointer in each object record
  – Keeps overhead low

• If class structure changes infrequently
  – Build complete method tables at initialization & when class structure changes

• If running program can create new classes
  – Well, not all things can be efficient
  – See Deutsch & Schiffman, POPL 1984
What is this JIT thing?

Just-in-time compilers

• JITs were developed to speed the execution of OOLs
  – Execution model is to interpret until “enough” † is known, & then compile
    → JITs capitalize on context that is discovered at runtime
  – Compiler runs during program execution
    → Compiler must, itself, be quite fast
    → Compiled code must be fast enough to cover cost of JIT execution
    → Creates an interesting set of design tradeoffs for the JIT
  – JITs have since been applied to many situations other than OOL execution

• First well-known JIT was for Smalltalk 80
  – Allowed $10,000 SUN 2 to compete with $180,000 Dorado

• Most Java systems now include a JIT
  – HotSpot Server Compiler is an excellent example
  – Dynamo was a bolt-on JIT for ordinary executables

† “Enough” covers a lot of ground, from the structure of the class hierarchy & the types of objects to knowledge of which methods are frequently invoked.