Runtime Support for OOLs

*Object Records, Code Vectors, Inheritance*

Comp 412
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
  – Name mangled labels plus access links or display for procedure local names

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
  – Need to find both data members and code members

Compiler must emit code that builds & maintains the runtime structures for addressability

This lecture focuses on runtime support for OOLs.
To compile method \( M \) of object \( O \) in class \( C \), the compiler needs:

1. Symbol table for current block and a search path that models the lexically scoped environment in \( M \)
   - Just like an \textbf{ALL} — inner declarations hide outer declarations

2. Set of symbol tables and a search path for the inheritance hierarchy
   - Class \( C \) and all of its superclasses
   - Need to find methods and instance variables in any superclass

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

\textit{Three search paths are needed for name resolution. In an ALL, the compiler could combine 1 & 3 for a single unified search. In Java, the compiler also needs a search path to model the inheritance hierarchy.}
From the lecture on naming

Compile-time Structures for OOLs — Java

Conceptually

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

The Java Name Space

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

Class ColorPoint extends Point {
    Color c;
    public void draw() {...} // override (hide) Point's draw
    public void test() { y = x; draw(); }
}

Class C {
    int x, y;
    public void m() {
        Point p = new ColorPoint(); // uses ColorPoint, and, by inheritance
        y = p.x;
        p.draw(); // the definitions from Point
    }
}

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 about OR pointers, similar to AR pointers
    → We won’t call them ORPs
• Classes are objects, too
  – ORs of classes instantiate the class hierarchy

Object Records

• Distinct storage for data members
• Fast, consistent access cost
  – Known constant offsets from OR pointer
• Linguistic provision for initialization

Max is a data member.
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

Data\_1 \& data\_2 are class data members.

This discussion is generic rather than Java-specific.

Could place code pointer at offset 0 to eliminate an add ...
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 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 memory access to find code for a function

This discussion is generic rather than Java-specific.
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 Implementation

- Single copy of the code for each method
- Each object could instantiate the code vector
  - Even faster access
  - More space
  - More work at object creation

Fundamental engineering work on OOLs was done in “small memory” days
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 (\textit{super-super}, \ldots)
- Subclass can add new members
- Subclass can override inherited definitions
  - ColorPoint inherits \( x \) & \( y \) from Point, adds \( m \), and overrides \textit{draw}()

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

- This feature is called \textbf{multiple inheritance}
- Can introduce conceptual problems with name resolution
- Languages use different rules to disambiguate these situations

COMP 412, Fall 2019
Representing Inheritance at Runtime

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 \( p.class \rightarrow Point.code + offset \rightarrow p.draw \)
- Hierarchy must be built (statically, at initialization, or at class load)

Code vectors are shown in the class rather than the object. Drawing gets more complex with copies of the code vector pointer in each object. The mechanics do not change (that much).
Inheritance has a strong impact on OR Layout

- OR needs to represent for each declared member (*code & data*), all the way up the class hierarchy (*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

```java
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.
Runtime Resolution for Code Members

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

Closed Class Structure

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

• Virtual functions force runtime resolution

Often called “function dispatch”
How does the running program find the code for a given method call?

Closed Class Structure

- Mapping of names to methods is static and known (C++)
  - Fixed offsets & indirect calls
  - `this \rightarrow class \rightarrow code vector \rightarrow function pointer`

- Virtual functions force runtime resolution

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

\[
\begin{align*}
\text{p is a Point} \\
\text{cp is a ColorPoint}
\end{align*}
\]

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

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

Open Class Structure

• Dynamic mapping, unknown until runtime
• In restricted cases, can use some more complex map built at runtime
  – Rebuild class vectors when class definitions & hierarchy change (?)
  – Use some dynamic structure based on hashing (?)

<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 call?

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 (truly slow!)

```
Class
+------------------------+
| class superclass code  |
|                         |
|                         |

Point
+------------------------+
| class superclass code  |
| draw                   |
| _._.Point.draw         |
|                         |

ColorPoint
+------------------------+
| class superclass code  |
| draw                   |
| _._.CP.draw            |
|                         |

Smalltalk 80
```
Runtime Resolution for Code Members

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

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.
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 \((\text{truly slow!})\)

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). \((\text{efficiency?})\)

**General problem:** updating ORs of instantiated objects when their class changes \((\text{language design issue})\)
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
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: \textit{this} \rightarrow \textit{class} \rightarrow \textit{code vector} (or \textit{this} \rightarrow \textit{code vector})
  – Call overhead is higher, in the best case
• Rest of pre-call, post-return, prolog, & epilog are as in an \textbf{ALL}

\textbf{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 \(<\textit{search key, class, method pointer}>\)

\textbf{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
  - But, you can do a lot to help (e.g., method lookup caches)
Method Lookup Caches

**A Method Lookup Cache retains method id to code translations**
- Cache maps method id to a single `<class, function pointer>` pair

**A global cache is a single, large, single-probe, software-run cache**
- Deutsch & Schiffman report 85 to 90% hit ratio
- 20 to 30% speedup above un-cached lookups
- Easy to implement; achieves strong results; memory cost is small

**An inline method cache has a single-entry cache at each call site**
- Stores class and function pointer from last time the call site executed
- If class changes, code performs the full lookup and updates inline cache
- Deutsch & Schiffman report 95% hit ratio
- 9 to 11% speedup above global cache

**Why an inline cache?**
- Obviates the need to name call sites
- Provides one cache per calling context
- Placing the data inline simplifies the lookup (PC-relative address)
Method Lookup Caches

**Inline method cache is easy to implement**

- The call works by checking the receiver’s class against the stored class
  - If they match, it retrieves the function pointer from the inline cache
  - If they do not match, it invokes the full lookup & updates the inline cache
- To start the process, initialize all the caches to `<invalid, invalid>`
- First call at that site will fail the cache lookup
- It then invokes the full lookup and links it into the cache
  - Stores class and function pointer into the inline cache
  - A good use for self-modifying code

**Why does this work well?**

- It capitalizes on class locality at each call site
- If the last call had an integer receiver, it is likely that this call will, too
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 (yes, a long time ago)
  - 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.