Naming in OOLs and Storage Layout

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.

Chapters 4, 5, 6 & 7 in EaC2e
What Are The Issues?

**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

*As stated earlier, this lecture focuses on the compile-time mechanisms. Runtime support for ALL & OOL name spaces will appear later.*
The Java Name Space

```java
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 { // uses ColorPoint
    int x, y;
    public void m()
    {
        Point cp = new ColorPoint();
        y = cp.x;
        cp.draw();
    }
}
```

In the example, Point cp is assigned a new ColorPoint, so it reference’s ColorPoint’s draw() code.

Point and ColorPoint have different runtime representations. The compiler needs:

- Names, types, and locations for members
  — Location is an offset from cp
- A mechanism to ensure that a call to draw() reaches the right instance
The Java Name Space

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 { // uses ColorPoint
    int x, y;
    public void m() {
        Point cp = new ColorPoint();
        y = cp.x;
        cp.draw();
    }
}

- Every object has an object record, including each class
- Items shared amongst a class are recorded in the class’ object record

The implementation is more complex, but that’s a story for another day.
Name Resolution in an OOL

What names can an executing code member (a function) access?

• Names defined by the code member
  – *And its surrounding lexical context*
• The receiving object’s data members
  – Smalltalk terminology: *instance variables*
• The code & data members of the class that defines it
  – *And its context from inheritance*
  – Smalltalk terminology: *class variables and methods*
• Any object defined in the global name space

The method might need the address for any or all of these objects

An OOL resembles an ALL, with a different name space

• In an ALL, scoping is relative to *hierarchy of the code*
• In an OLL, scoping is relative to *hierarchy of both the code & the data*
Concrete Example: The Java Name Space

**Code within a method M for object O of class C can see:**

- Local variables declared within M (*standard lexical scoping*)
- All instance variables of O & class variables of C
- All public and protected variables of any *superclass* of C
- Classes defined in the same package as C or in any explicitly imported package
  - public class variables and public instance variables of imported classes
  - package class and instance variables in the package containing C
- Class declarations can be nested!
  - Member declarations hide outer class declarations of the same name
  - Accessibility options: public, private, protected, package

Both lexical nesting & class hierarchy at play

*Superclass* is an ancestor in the inheritance hierarchy
Compile-time Structures for OOLs — Java

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
   - Just like 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 can 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.
Again, the “sheaf of tables” implementation simplifies the conceptual picture.
Static coordinate needs a hierarchy field, as well.
Compile-time Structures for OOLs — Java

To find the address for a reference to $x$ in method $M$ for an object $O$ of class $C$, the compiler must:

• For an unqualified use (i.e., $x$):
  – Search the symbol table for the method’s lexical hierarchy
  – Search the symbol tables for the receiver’s class hierarchy
  – Search global symbol table (current package and imported)
  – For each hit, check visibility attribute of $x$

• For a qualified use (i.e.: $Q.x$):
  – Find $Q$ by the method above
  – Search from $Q$ for $x$
    – Must be a class or instance variable of $Q$ or some class it extends
  – Check visibility attribute of $x$

Compile-time cost increases by a small constant factor.
What About Storage Layout?

Where does all of this stuff go? And how does it get there?

The compiler must classify all code & data so that it can assign storage

- **Scopes**: local, global, subject to some set of lexical scoping constraints
- **Lifetimes**: entire execution, execution of a procedure, or indeterminate

Given these classifications & the state of the naming model, the compiler can assign data to specific data areas

- Each procedure has a local data area
  - *Local data from a scope smaller than a procedure usually goes here*
- Declarations, procedures, files, & modules may have a static data area
  - *Depends on the specific declarations in the code*
- A program may have one or more global data areas & constant pools
- Each object has an object record
  - *A class, being an object, has an object record of class “class”*
When Does the Compiler Assign Storage?

The compiler must assign storage before using addresses in the IR

- Code contains implicit information about storage
- Compiler must make its decisions after parsing declarations
  - Can envision batch schemes or incremental schemes
  - Either way, the compiler assigns addresses

Without a “declarations before statements” rule, the compiler must either build an IR that abstracts away addressing, or make two passes.
When Does the Compiler Assign Storage?

The compiler must assign storage before using addresses in the IR

- Code contains implicit information about storage
- Compiler must make its decisions after parsing declarations
  - Can envision batch schemes or incremental schemes
  - Either way, the compiler assigns addresses

```
Procedure → Header { Decls StmtList }
Decls → DeclList
DeclList → DeclList Declaration
  | Declaration
StmtList → StmtList Statement
  | Statement
```

Modify the grammar to create a reduction to assign storage.

Typical Grammar for a Procedure

Final points:
1. When does this happen? At compile time
2. Where does stuff go? Next lecture
Translating Names

How does the compiler represent a specific instance of a variable \( x \)?

- Name is translated into a static coordinate
  - \(< \text{level}, \text{offset} >\) pair
    - “level” is lexical nesting level of the declaration
    - “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

Global names are in a table for the global name space, with its own offset — say negative one.
The compiler must assign a location to each named variable and each temporary value that it introduces? Where do all these values live?

• The answer should depend on the value’s visibility & its lifetime

Lifetimes

Automatic variables

• Implicit allocation & deallocation
• Lifetime matches lifetime of declaring scope’s activation

Static variables

• Implicit allocation and deallocation
• Lifetime is the complete execution of the program

Values with irregular lifetimes

• Explicit allocation and deallocation (new, or malloc, or ...)

Global variables are static, in almost any PL.
Storage Layout

The compiler must assign a location to each named variable and each temporary value that it introduces? Where do all these values live?

• The answer should depend on the value’s visibility & its lifetime

Visibility

Local Scopes

• Procedure scope: name is visible in defining procedure (& nested ones)
• Block scope (C, scheme): name is visible in defining block (& nested ones)

Global Scope

• Names are visible anywhere that they are declared, unless obscured by a local declaration

Unusual scopes

• C’s “file scope”: a variable declared global & static is visible throughout the file where it is declared, but nowhere else
Where do all these values live? AR refers to a procedure’s activation record

**Automatic & Local**
- Automatic local values live in the local data area, in the procedure’s AR*
  → *Or in a register, if it is an unambiguous scalar value*
- Compiler lays out a map for the local data area, including spill locations
- Local values from surrounding scopes live in the AR for their defining scope
  → *Runtime addressability mechanism establishes a way to reach them*

**Static** *(implies lifetime of entire execution)*
- Procedure static ⇒ procedure-specific data area (e.g., &_.procname_da)
- File static ⇒ file-specific data area (e.g., &_.filename_da)
- Necessary to ensure that they remain live across different activations

**Global** *(implies lifetime of entire execution)*
- Global values live in one or more global data area(s)
- One label per variable, or per file, or per program, ...
  → *With consistently mangled labels to connect them across compiles*
  → *Equivalent to an assembly language “defined storage” pseudo-operation*

Compilers obtain these labels by “mangling” the original name.
The linker will report conflicting definitions of the same label.
Activation Record Basics

In most systems, Activation Records have a similar layout

- **space for parameters**: Space for parameters to the current procedure
- **register save area**: Contents of saved registers
- **slot for return value**: If function, space for return value
- **slot for return address**: Address to resume caller
- **slot for addressability**: Help with non-local access
- **slot for caller’s ARP**: To restore caller’s ARP on return
- **space for local variables**: Space for local variables, temporaries, & spills

ARP ≅ Activation Record Pointer

What is this “activation record”?

One AR for each invocation of a procedure
One register dedicated to hold the current ARP
Name Mangling

One of the most useful tools the compiler has is an assembly-code label
• Compiler can emit a label in the code
• Assembler will turn it into a relocatable symbol
• Linker will replace the label with the appropriate virtual address
• Hardware will translate virtual address to a physical address

Name mangling is the standard trick
• The compiler assumes that some source-language names are unique
  – Global variables, procedure names
  – Use fully-qualified names for nested objects
• The compiler builds strings based on the unique name
  – Code for procedure \texttt{fee} might be labelled \_fee\_ or \&_fee or ... 
  – Specific “mangles” for specific purposes
    → Static data area, entry point to the code, return address at a call, ...
  – Use character combinations that are not legal in the source language
Alignment Issues: One More Standard Trick

Having values of multiple types in the same data area creates potential alignment issues

- Each type (size) of value has its own alignment restriction
- Laying out a data area in arbitrary order can require padding
  - Consider \(a\) & \(c\) as single-byte characters and \(b\) & \(d\) as single-word integers

A Layout That Wastes Space

A Better Layout
Alignment Issues: One More Standard Trick

To create a layout that minimizes space wasted in “padding”

• Sort variables by alignment restrictions
  – Most strict (e.g., quad word or double word) to least strict (e.g., byte)

• From most strict to least strict, lay out variables in contiguous memory
  – Alignment constraints generally decrease by ½ at a transition
  – This scheme avoids almost all padding for scalar variables.
  – Structures that have atypical alignment constraints may need padding

  → A seventeen-byte long structure followed by a quadword aligned double double

<table>
<thead>
<tr>
<th>b</th>
<th>d</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
</tbody>
</table>

whole words | bytes | unused memory
Where Do These Data Areas Go?

The compiler lays out data areas
- Where are they located at runtime?
- How does the code find the data areas?

The executable assumes that it has its own virtual address space
- Large, isolated, uninitialized tract of memory
- Compiled code manages the layout of that virtual address space
  - In collaboration with the operating system, which creates & maintains it

Virtual Address Space

| 0 | 1 | 2 | 3 | ... | n-2 | n-1 | n |
Address Space Layout

Most language runtimes layout the address space in a similar way

- Pieces (stack, heap, code, & globals) may move, but all will be there
- Stack and heap grow toward each other (if heap grows)
- Arrays live on one of the stacks, in the global area, or in the heap

The picture shows one virtual address space.
- The hardware supports one virtual address space per process.
- How does a virtual address space map into physical memory?
Multiple Virtual Address Spaces?

The Big Picture

Compiler’s view

virtual address spaces (one per process)

OS’ view

1980 Hardware view

Physical address space (big vector of memory)

TLB is an address cache used by the OS to speed virtual-to-physical address translation. A processor may have > 1 level of TLB.
Unified address space

TLB is an address cache used by the OS to speed virtual-to-physical address translation. A processor may have > 1 level of TLB.

Mapping Virtual Address Spaces

Add a couple of high-end bits so we can treat the process address spaces as one large address space for mapping purposes.

TLB

OS’ view

1980 Hardware view

Physical address space (big vector of memory)
Of course, the “Hardware view” is no longer that simple

- Multiple levels of cache
- Caches shared among cores
- Caches exclusive to one core
- Multiple levels of (shared) TLB

All of the addresses must map in a way that works for the code generated to run in a single virtual address space

Most processors have > 1 core