Implementing Object-Oriented Languages
- Part 1

Y.N. Srikant
Computer Science and Automation
Indian Institute of Science
Bangalore 560 012

NPTEL Course on Compiler Design
Outline of the Lecture

- Language requirements
- Mapping names to methods
- Variable name visibility
- Code generation for methods
- Simple optimizations

Parts of this lecture are based on the book, “Engineering a Compiler”, by Keith Cooper and Linda Torczon, Morgan Kaufmann, 2004, sections 6.3.3 and 7.10.
Language Requirements

- Objects and Classes
- Inheritance, subclasses and superclasses
- Inheritance requires that a subclass have all the instance variables specified by its superclass
  - Necessary for superclass methods to work with subclass objects
- If A is B’s superclass, then some or all of A’s methods/instance variables may be redefined in B
Example of Class Hierarchy with Complete Method Tables
Mapping Names to Methods

- Method invocations are not static calls
- \texttt{a.fee()} invokes \texttt{one.fee()}, \texttt{b.foe()} invokes \texttt{two.foe()}, and \texttt{c.fum()} invokes \texttt{three.fum()}
- Conceptually, method lookup behaves as if it performs a search for each procedure call
  - These are called virtual calls
  - Search for the method in the receiver’s class; if it fails, move up to the receiver’s superclass, and further
  - To make this search efficient, an implementation places a complete method table in each class
  - Or, a pointer to the method table is included (virtual tbl ptr)
Mapping Names to Methods

- If the class structure can be determined wholly at compile time, then the method tables can be statically built for each class.

- If classes can be created at run-time or loaded dynamically (class definition can change too):
  - Full lookup in the class hierarchy can be performed at run-time or
  - Use complete method tables as before, and include a mechanism to update them when needed.
Rules for Variable Name Visibility

- Invoking `b.fee()` allows `fee()` access to all of `b`’s instance variables `(x,y,z)`, (since `fee` and `b` are both declared by class one), and also all class variables of classes one, two, and three.

- However, invoking `b.foe()` allows `foe()` access only to instance variables `x` and `y` of `b` (not `z`), since `foe()` is declared by class two, and `b` by class one.
  - `foe()` can also access class variables of classes two and three, but not class variables of class one.
Example of Class Hierarchy with Complete Method Tables
Methods can access any data member of any object that becomes its receiver
  - receiver - every object that can find the method
  - subject to class hierarchy restrictions

Compiler must establish an offset for each data member that applies uniformly to every receiver

The compiler constructs these offsets as it processes the declarations for a class
  - Objects contain no code, only data
Single Class, No Inheritance

Example:
Class giant {
    int fee() {...}
    int fie() {...}
    int foe() {...}
    int fum() {...}
    static n;
    int x,y;
}

Class record

Method pointer offset

giant

joe

fred

Object layout

x

y

Method pointer offset

%giant.new_
%giant.fee_
%giant.fie_
%giant.foe_
%giant.fum_

2

13

14

15

16

x

y

Y.N. Srikant
Implementing Single Inheritance

Diagram showing class inheritance relationships with object and method details.
# Single Inheritance Object Layout

<table>
<thead>
<tr>
<th>Object layout for joe/fred (giant)</th>
<th>class pointer</th>
<th>sc data members (x)</th>
<th>mc data members (y)</th>
<th>giant data members (z)</th>
</tr>
</thead>
<tbody>
<tr>
<td>class record for class giant</td>
<td>class pointer</td>
<td>superclass pointer</td>
<td>%new_pointer</td>
<td>%fee_pointer</td>
</tr>
<tr>
<td>Object layout for goose (mc)</td>
<td>class pointer</td>
<td>sc data members (x)</td>
<td>mc data members (y)</td>
<td></td>
</tr>
<tr>
<td>class record for class mc</td>
<td>class pointer</td>
<td>superclass pointer</td>
<td>%new_pointer</td>
<td>%fee_pointer</td>
</tr>
<tr>
<td>Object layout for jack (sc)</td>
<td>class pointer</td>
<td>sc data members (x)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>class record for class sc</td>
<td>class pointer</td>
<td>superclass pointer</td>
<td>%new_pointer</td>
<td>%fee_pointer</td>
</tr>
</tbody>
</table>

Y.N. Srikant
Now, an instance variable has the **same offset** in every class where it exists up in its superclass.

Method tables also follow a similar sequence as above.

When a class redefines a method defined in one of its superclasses:
- the method pointer for that method implementation must be stored at the same offset as the previous implementation of that method in the superclasses.
Implementing Multiple Inheritance

- Assume that class c inherits from classes a and b, but that a and b are unrelated in the inheritance hierarchy
- Assume that class c implements fee, inherits fie from a, and inherits foe and fum from b
- The class diagram and the object layouts are shown next
Implementing Multiple Inheritance

class a

%new_pointer

fie ...

%new_pointer

class b

%new_pointer

foe ...

fum ...

class c

%new_pointer

fee ...

Implementing Multiple Inheritance

<table>
<thead>
<tr>
<th>Object Layout for Objects of Class A</th>
<th>Class</th>
<th>A Data Members</th>
</tr>
</thead>
<tbody>
<tr>
<td>object layout for objects of class a</td>
<td>class pointer</td>
<td>a data members</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Object Layout for Objects of Class B</th>
<th>Class</th>
<th>B Data Members</th>
</tr>
</thead>
<tbody>
<tr>
<td>object layout for objects of class b</td>
<td>class pointer</td>
<td>b data members</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Object Layout for Objects of Class C</th>
<th>Class</th>
<th>A Data Members</th>
<th>B Data Members</th>
<th>C Data Members</th>
</tr>
</thead>
<tbody>
<tr>
<td>object layout for objects of class c</td>
<td>class pointer</td>
<td>a data members</td>
<td>b data members</td>
<td>c data members</td>
</tr>
</tbody>
</table>
Implementing Multiple Inheritance

When `c.fie()` (inherited from `a`) is invoked with an object layout as above, it finds all of `a`’s instance variables at the correct offsets

- Since `fie` was compiled with class `a`, it will not (and cannot) access the other instance variables present in the object and hence works correctly

Similarly, `c.fee()` also works correctly (implemented in `c`)

- `fee` finds all the instance variables at the correct offsets since it was compiled with class `c` with a knowledge of the entire class hierarchy
Implementing Multiple Inheritance

class a
%new_pointer

fie ...

class c
%new_pointer

class b
%new_pointer

foe ...

fum ...

fee ...
Implementing Multiple Inheritance

However, invoking `c.foe()` or `c.fum()` creates a problem

- `foe` and `fum` are inherited from `b`, but invoked from an object of class `c`
- Instance variables of class `b` are in the wrong place in this object record – sandwiched between the instance variables of classes `a` and `c`
- In objects of class `b`, the instance variables are at the start of the object record
- Hence the offset to the instance variables of class `b` inside an object record of class `c` is unknown
Implementing Multiple Inheritance

- To compensate for this, the compiler must insert code to adjust the receiver pointer so that it points into the middle of the object record – to the beginning of b’s instance variables.
- There are two ways of doing this.
Implementing Multiple Inheritance
- Fixed Offset Method

- Record the constant offset in the method table along with the methods
  - Offsets for this example are as follows:
    - (c) fee : 0, (a) fie: 0, (b) foe : 8, (b) fum : 8, assuming that instance variables of class \texttt{a} take 8 bytes
    - Generated code adds this offset to the receiver’s pointer address before invoking the method
Implementing Multiple Inheritance

class a

%new_pointer

fie ...

class b

%new_pointer

foe ...

class c

%new_pointer

fie ...

Object layout for objects of class c

class pointer

a data members

b data members

c data members

data pointer for c.foe()
Implementing Multiple Inheritance
- Trampoline Functions

- Create **trampoline** functions for each method of class \textit{b}
  - A function that increments \textit{this} (pointer to receiver) by the required offset and then invokes the actual method from \textit{b}.
  - On return, it decrements the receiver pointer, if it was passed by reference
Implementing Multiple Inheritance

- Trampolines appear to be more expensive than the fixed offset method, but not really so
  - They are used only for calls to methods inherited from \b
    - In the other method, offset (possibly 0) was added for all calls
  - Method inlining will make it better than option 1, since the offset is a constant
- Finally, a duplicate class pointer (pointing to class \c) may need to be inserted just before instance variables of \b (for convenience)