Binding and Scoping

Read: Scott, Chapter 3.1, 3.2 and 3.3.1, 3.3.2 and 3.3.6
Lecture Outline

- Notion of binding time
- Object lifetime and storage management
- An aside: Stack Smashing 101

- Scoping
  - Static scoping
  - Dynamic scoping
Notion of Binding Time

- **Binding time** (Scott): the time an answer becomes associated to an open question
Notion of Binding Time

- **Static**
  - Before program execution

- **Dynamic**
  - During program executes
Examples of Binding Time Decisions

- **Binding time** (Scott): the time an answer becomes associated to an open question

- Binding a variable name to a memory location
  - Static or dynamic
  - Determined by **scoping rules**

- Binding a variable/expression to a type
  - Static or dynamic

- Binding a call to a target subroutine
  - Static (as it is in C, mostly) or dynamic (virtual calls in Java, C++)
Example: Binding Variables to Locations

- Map a variable to a location
  - Map variable at \texttt{use} to a location
  - Map subroutine at \texttt{use} to target subroutine
- Determined by \texttt{scoping rules}
  - Static scoping
    - Binding before execution
  - Dynamic scoping
    - Binding during execution
- More on scoping later…

```c
int x,y;
void foo(int x)
{
    y = x;
    int y = 0;
    if (y) {
        int y;
        y = 1;
    }
}
```
General View of Dynamic Binding

- Dynamic binding
  - What are the advantages of dynamic binding?
  - Disadvantages?

- An example: Cost of dynamic binding of call to target method in OO languages
Example: Cost of Dynamic Dispatch in C++

- Source: Driesen and Hölzle, OOPSLA’ 96

Virtual function tables (VFTs)
Capital characters denote classes, lowercase characters message selectors, and numbers method addresses

- load [object_reg+#VFTOffset],table_reg
- load [table_reg+#selectorOffset],method_reg
- call method_reg

Extra instructions: cost extra
Other Choices Related to Binding Time

- Pointers: introduce “heap variables”
  - Good for flexibility – allows dynamic structures
  - Bad for efficiency – direct cost: accessed indirectly; indirect cost: compiler unable to perform optimizations

- Most PLs support pointers
  - Issues of management of heap memory
    - Explicit allocation and deallocation
    - Implicit deallocation (garbage collection)

- PL design choices – many subtle variations
  - No pointers (FORTRAN 77)
  - Explicit pointers (C++ and C)
  - Implicit pointers (Java)
Lecture Outline

- Notion of binding time
- Object lifetime and storage management

- An aside: Stack Smashing 101

- Scoping
  - Static scoping
  - Dynamic scoping
Storage Allocation Mechanisms

- **Static storage** – an object is given absolute address, which is the same throughout execution
  - What is an example of static data?

- **Stack storage** – stack objects are allocated on a run-time stack at subroutine call and deallocated at return
  - Needs a stack management algorithm
  - What is an example of stack data?

- **Heap storage** - long-lived objects are allocated and deallocated at arbitrary times during execution
  - Needs the most complex storage management algorithm
Combined View

- **Static storage**: `.text` (program code), `.rodata`, `.data`, etc.

- **Stack** contains one **stack frame** per executing subroutine
  - Stack grows from higher towards lower memory addresses

- **Heap** contains objects allocated and not yet de-allocated
  - Heap grows from lower towards higher memory addresses

Memory graph courtesy of RPISEC/MBE class.
Examples of Static Data

- Program code
- Global variables
- Tables of type data (e.g., inheritance structure)
- Dispatch tables (VFTs) and other tables
- Other
Examples of Stack Data

- What data is stored on the stack?
- Local variables, including parameters
- Compiler-generated temporaries (i.e., for expression evaluation)
- Bookkeeping (stack management) information
- Return address
Run-time Stack

- Stack contains frames of all subroutines that have been entered and not yet exited from.
- Frame contains all information necessary to update stack when subroutine is exited.
- Stack management uses two pointers: \texttt{fp (frame pointer)} and \texttt{sp (stack pointer)}.
  - \texttt{fp} points to a location at the start of current frame.
    - In higher memory (but lower on picture).
  - \texttt{sp} points to the next available location on stack (or the last used location on some machines).
    - In lower memory (but higher up on picture).
  - \texttt{fp} and \texttt{sp} define the beginning and the end of the frame.
Run-time Stack
Run-time Stack Management

- Addresses for local variables are encoded as $sp + \text{offset}$
  - But may also have $fp - \text{offset}$

- Idea:
  - When subroutine is entered, its frame is placed on the stack. $sp$ and $fp$ are updated accordingly
  - All local variable accesses refer to this frame
  - When subroutine is exited, its frame is removed from the stack and $sp$ and $fp$ are updated accordingly
Frame Details

- Arguments to called routines
- Local variables, including parameters
- Temporaries

- Miscellaneous bookkeeping information
  - Saved address of start of caller’s frame (old fp)
  - Saved state (register values of caller), other

- Return address
Frame Example

```c
void foo(double rate, double initial) {
    double position; ...
    position = initial + rate*60.0; ...
    return;
}
```

Assume `bar` calls `foo`.

Frame for `foo`:

<table>
<thead>
<tr>
<th>sp</th>
<th>position</th>
<th>initial</th>
<th>rate</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fp</td>
<td>old fp</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>return address</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Locals</th>
<th>Temporaries</th>
<th>Misc bookkeeping info</th>
<th>Return address in code of caller</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
